Looped Transformers as Programmable Computers

Anonymous Authors[1]

Abstract

We present a framework for using transformer networks as universal computers by programming them with specifsic weights and placing them in a loop. Our input sequence acts as a punchcard, consisting of instructions and memory for data read/writes. We demonstrate that a constant number of encoder layers can emulate basic computing blocks, including lexicographic operations, non-linear functions, function calls, program counters, and conditional branches. Using this framework, we emulate a computer using a simple instruction-set architecture, which allows us to map iterative algorithms to programs that can be executed by a constant depth looped transformer network. We show how a single frozen transformer, instructed by its input, can emulate a basic calculator, a basic linear algebra library, and even a full backpropagation, in-context learning algorithm. Our findings reveal the potential of transformer networks as programmable compute units and offer insight into the mechanics of attention. An implementation is given in this link.

1 Introduction

Transformers (TFs) have become a popular choice for machine learning tasks, achieving state-of-the-art results in Natural Language Processing (NLP) and Computer Vision (CV) (Vaswani et al., 2017; Khan et al., 2022; Yuan et al., 2021; Dosovitskiy et al., 2020). They use attention to capture higher-order relationships and long-range dependencies, making them effective in tasks such as machine translation and language modeling (Vaswani et al., 2017; Kenton & Toutanova, 2019). Large language models (LLMs) such as GPT-3 (Brown et al., 2020) and PaLM (Chowdhery et al., 2022), with billions of parameters, have achieved state-of-the-art performance on many NLP tasks. These models can also perform in-context learning (ICL), adapting to and performing a specific task based on a brief prompt and a few examples.

LLMs can also perform algorithmic tasks and reasoning through ICL, as shown in several works, such as (Nye et al., 2021; Wei et al., 2022c; Lewkowycz et al., 2022; Wei et al., 2022b; Dasgupta et al., 2022; Chung et al., 2022). For example, (Zhou et al., 2022) showed that LLMs can perform addition on unseen examples when prompted with a multi-digit addition algorithm and some examples. These results suggest that LLMs can apply algorithmic principles and perform pre-instructed commands on a given input, as if interpreting natural language as code.

Transformers can simulate Turing Machines with sufficient depth or recursive links around attention layers (Pérez et al., 2021; Pérez et al., 2019; Wei et al., 2022a). These constructions do not provide specific guidance on constructing TFs that perform specific algorithmic tasks. However, specialized designs can allow TFs to compile programs in a higher level programming language or execute higher level programs. For example, in (Weiss et al., 2021), a computational model and a programming language was designed that maps simple selection and aggregation commands on indexed input tokens, which can be used to create several interesting algorithms. Programs written in Restricted Access Sequence Processing Language (RASP) can then be mapped into transformer networks, which typically scale in size with the size of the program.

Recently, various methods have been developed to select the weights of a Transformer model to function as a learning algorithm on-the-fly, performing implicit training at inference time when given training data as input (Akyürek et al., 2022; von Oswald et al., 2022). These methods typically require a number of layers proportional to the number of iterations of the learning algorithm and are limited to a small set of loss functions and models.

Our paper aims to explore what algorithms can transformer networks efficiently emulate (*i.e.*, within small depth/width) at inference-time and present our contributions towards understanding the capabilities of transformer networks as programmable computers.

Our Contributions: In this paper, we show that transformer networks can emulate complex algorithms and programs by programming them with specific weights and placing them in a loop. We accomplish this by reverse engineering attention to emulate basic computing blocks, such as lexicographic operations, nonlinear functions, function calls, program counters and conditional branches. We also

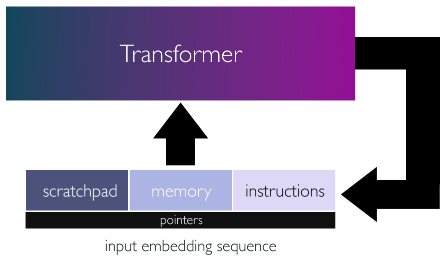


Figure 1. A sketch of the looped transformer architecture, where the input sequence stores the commands, memory where the data is read/written from, and a scratchpad where intermediate results are stored. The input is processed by the network and the output is used as the new input, allowing the network to iteratively update an implicit state and perform complex computations.

demonstrate the importance of using a single loop or recursion to connect the transformer's output sequence back to its input, avoiding the need for a deep model.

We design a transformer that can execute programs written in a generalized version of a single instruction, known as `SUBLEQ`(A,B,C), which is a one-instruction set computer (OISC) that consists of 3 memory address operands. When executed, it subtracts the value at memory address A from the value at memory address B and stores the result in B. If the result in B is less than or equal to zero, the execution jumps to address C, otherwise it proceeds to the next instruction. Programs written in `SUBLEQ` language use only this command, yet this single instruction is capable of defining a universal computer (Mavaddat & Parhami, 1988).

We construct transformers that can run programs like `SUBLEQ` using a more flexible instruction called `FLEQ` with

$$\begin{aligned}&\text{mem}[c] = f_m(\text{mem}[a], \text{mem}[b])\\ &\texttt{if mem}[\text{flag}] \leq 0 \;\; \texttt{goto} \text{ instruction } p\end{aligned}$$

format, where f_m can be selected from a set of functions (matrix multiplication/ non-linear functions/ polynomials/ etc), which we can hardcode into the network. The depth of the transformer needed to run these programs is not affected by the program's complexity, but by the depth required for a single `FLEQ` instruction, which is typically constant. We use this framework to emulate a calculator, linear algebra functions and in-context learning algorithm. The input sequence acts as a program for the transformer to execute, while also providing space to store and process variables. The transformer networks used to execute these programs have a depth of 13 or less.

Our study shows that attention mechanisms can be used to emulate complex iterative algorithms and execute general programs with even a single loop. We hope that this inspires more research on the capabilities of attention and the use of smaller transformer networks to distill tasks for larger models and enhance language model capabilities.

2 Prior Work

Our work is inspired by the recent results on the expressive power of Transformer networks and their in-context learning capabilities. The authors of Pérez et al. (2021); Pérez et al. (2019); Wei et al. (2022a) have shown that Transformers are Turing complete, meaning they can simulate a Turing machine. The constructions typically require high/infinite precision (apart from that of (Wei et al., 2022a)), and recursion around attention layers. Additionally, Yun et al. (2019) prove that with sufficient width/depth, Transformers can act as universal sequence to sequence approximators. In (Weiss et al., 2021), the authors propose a computational model for the transformer-encoder in the form of a domain-specific language called the Restricted Access Sequence Processing Language (RASP). The model maps the basic components of a TF encoder into simple primitives. Examples of tasks that could be learned by a Transformer are provided, and the maximum number of heads and layers necessary to encode a task in a transformer are analyzed.

In a recent and related work, (Lindner et al., 2023) suggests using transformer networks as programmable units and introduces a compiler called Tracr which utilizes RASP. However, the expressivity limitations and unclear Turing completeness of the language are discussed in (Weiss et al., 2021; Merrill et al., 2022; Lindner et al., 2023). Our approach, in contrast, demonstrates the potential of transformer networks to serve as universal computers, enabling the implementation of arbitrary nonlinear functions and emulating iterative, non-linear algorithms. Furthermore, our framework allows the depth of our transformers *to not* scale in proportion to the lines of code that they execute, allowing the implementation of iterative algorithms, expanding the potential applications.

In (Garg et al., 2022) the authors demonstrate that standard Transformers (*e.g.*, GPT-2) can be trained from scratch to perform in-context learning of linear functions and more complex model classes, such as two-layer neural networks, with performance that matches or exceeds task-specific learning algorithms. An element useful to our derivation is the fact that language is completely removed from the picture, and they perform all operations on the level of vector embeddings.

Motivated by the above experimental work, in (Akyürek et al., 2022), the authors investigate the hypothesis that TF-based in-context learners emulate standard learning algorithms implicitly at inference time. The authors provide evidence for this hypothesis by constructing transformers that implement SGD for linear models, showing that trained in-context learners closely match the predictors computed by these algorithms. It is

In a similar vein, (von Oswald et al., 2022) provides a hardcoded weight construction showing the equivalence between data transformations induced by a single linear self-attention layer and gradient descent on a regression loss. The authors empirically show that when training linear attention TFs on simple regression tasks, the models learned by GD and Transformers have intriguing similarities.

In (Liu et al., 2022), the authors test the hypothesis that TFs can perform algorithmic reasoning using fewer layers than the number of reasoning steps, in the context of finite automata. The authors characterized "shortcut solutions" that allow shallow Transformer models to exactly replicate the computation of an automaton on an input sequence, and showed that these solutions can be learned through standard training methods. As is expected this hypothesis is only true for a certain family of automata, as the general existence of shortcut solutions would imply the collapse of complexity classes that are widely believed not to be identical.

Other experimental studies have utilized recursion in transformer architectures in a similar manner to our constructions, although in our case we only utilize a single recursive link that feeds the output of the transformer back as an input (Hutchins et al., 2022; Shen et al., 2022; Dehghani et al., 2018). Relevant to our work is (Kirsch & Schmidhuber, 2021), in which the authors create a LSTM-based architecture that learns to perform backpropagation.

3 Preliminaries

The transformer architecture. Our work follows a similar problem setting as previous studies (e.g. (Yun et al., 2019; Garg et al., 2022; Akyürek et al., 2022; von Oswald et al., 2022)) in which the input sequence consists of d-dimensional embedding vectors rather than tokens. This simplifies our results without sacrificing generality, as an embedding layer can map tokens to the desired vector constructions.

The input to each layer, $\mathbf{X} \in \mathbb{R}^{d\times n}$, is a vector representation of a sequence of n tokens, where each token is a d-dimensional column. In this paper, the terms "token" and "column" may be used interchangeably. A transformer layer outputs $f(\mathbf{X})$, where f is defined as

$$\text{Attn}(\mathbf{X}) = \mathbf{X} + \sum_{i=1}^{H} \mathbf{V}^i \mathbf{X} \sigma_{\text{S}}(\mathbf{X}^\top \mathbf{K}^{i\top} \mathbf{Q}^i \mathbf{X}) \tag{1a}$$

$$f(\mathbf{X}) = \text{Attn}(\mathbf{X}) + \mathbf{W}_2 \sigma(\mathbf{W}_1 \text{Attn}(\mathbf{X}) + \mathbf{b}_1 \mathbf{1}_n^\top) + \mathbf{b}_2 \mathbf{1}_n^\top \tag{1b}$$

where σ_{S} is the softmax function applied on the columns of the input matrix, *i.e.*, $[\sigma_{\text{S}}(\mathbf{X}, \lambda)]_{i,j} = \frac{e^{\lambda X_{i,j}}}{\sum_{k=1}^{n} e^{\lambda X_{k,j}}}$, where $\lambda \geq 0$ is the temperature parameter, $\sigma(x) = x \cdot 1_{x>0}$ is the ReLU activation, and $\mathbf{1}_n$ is the all ones vector of length n. We refer to the $\mathbf{K}$, $\mathbf{Q}$, and $\mathbf{V}$ matrices as the key, query, and value matrices respectively[1] ; the superscript i that appears on the weight matrices indicates those corresponding to the i-th attention head.The first equation Equation (1a) represents the attention layer, while the combination with ReLUs a single transformer layer.

Iterative computation through a loop. In the following sections, we utilize TF networks with multiple transformer layers. Let us refer to the output of such a multilayer TF as $\mathsf{TF}(\mathbf{W}; \mathbf{X})$, where for simplicity $\mathbf{W}$ is the collection of all weight matrices required to define such a multi-layer TF. We use our constructions recursively, and feed the output back as an input sequence, allowing the network to perform iterative computation through a simple fixed-point like iteration. This recursive transformer is similar to past work on adding recursion to TF networks. We refer to these simple recursive TFs as *Looped Transformers*.

Algorithm 1 Looped Transformer

```
1: for $i = 1 : T$ do
2:     $\mathbf{X} \leftarrow \mathsf{TF}(\mathbf{W}; \mathbf{X})$
3: end for
```

This model is similar to how a traditional computer processes machine code, where it continually reads/writes data in memory, by executing one instruction at a time. The input sequence $\mathbf{X}$ includes the instructions and memory. Similar to how a CPU processes each line of code in a program, the transformer network processes parts of the input sequence to perform complex computationsand acts as a self-contained computational unit. The use of loops in this process is analogous to how CPUs operate using cycles.

While the analogy between TFs and CPUs can be entertaining, there are also many differences in implementation. It is important to keep these differences in mind and not rely too heavily on the analogy. The results obtained from using TFs as computational units do not require the analogy to be valid.

Input sequence format. The input to our transformer network has the following abstract form:

$$\mathbf{X} = \left[\begin{array}{ccc|ccc|ccc} & \mathbf{S} & & & \mathbf{M} & & & \mathbf{C} & \\ \mathbf{p}_1 & \cdots & \mathbf{p}_s & \mathbf{p}_{s+1} & \cdots & \mathbf{p}_{s+m} & \mathbf{p}_{s+m+1} & \cdots & \mathbf{p}_n \end{array}\right] \tag{2}$$

where $\mathbf{S}$ represents the portion of the input that serves as a "scratchpad," $\mathbf{M}$ represents the portion that acts as memory that can be read from and written to, and $\mathbf{C}$ represents the portion that contains the commands provided by the user. The $\mathbf{p}_1, \ldots, \mathbf{p}_n$ are positional encodings for the n columns, which will be described in more detail in the following paragraph, and will be used as pointers to data and instructions. The structure of our input sequence bares

[1]Typically the weight matrices are denoted as $\mathbf{W}_Q, \mathbf{W}_K, \mathbf{W}_V$ but to make notation cleaner, we use instead $\mathbf{Q}, \mathbf{K}, \mathbf{V}$.

similarities to that of (Wei et al., 2022a; Akyürek et al., 2022) that also use scratchspace, and have a separate part for the input data.

Scratchpad. The scratchpad is a crucial component of our constructions. This is the central location where the inputs and outputs of all computation are recorded. It is perhaps useful to think of this as an analogue to a CPU's cache memory. It functions as a temporary workspace where data is copied, transformed, and manipulated in order to perform a wide variety of operations, ranging from simple arithmetic to more complex tasks such as matrix inversion. The data necessary for the operation is always transferred from the memory to the scratchpad, and once the computation is completed, the data is transferred back to the memory.

Memory. All the compute boxes we create require memory to perform specific actions. The memory component of the input sequence serves as a storage location for data. This data can take various forms, including scalars, vectors, and matrices, and is subject to manipulation through various operations. As mentioned in the previous paragraph, the memory communicates with the scratchpad and serves as a central repository for all relevant data, allowing it to be accessed and manipulated as needed.

Commands. Our framework implements a set of commands within a transformer network; these serve as instructions that guide the internal functioning of the transformer, similar to a low-level programming language. These commands include indicators for memory locations and operation directives, allowing the TF to execute complex computations and tasks in a consecutive and organized manner.

4 Emulating a Generalized One-instruction Set Computer

4.1 A `SUBLEQ` Transformer

(Mavaddat & Parhami, 1988) showed that there exists an instruction such that any computer program can be translated to a program consisting of instantiation of this single instructions. A variant of such an instruction is `SUBLEQ`, where different registers, or memory locations are accessed. The way that `SUBLEQ` works is simple. It accesses two registers in memory, takes the difference of their contents and stores it back to one of the registers, and then if the result is negative it jumps to a different predefined line of code, or continues on the next instruction from the current line of code.[2] A computer that is built to execute `SUBLEQ` programs is called an One-Instruction Set Computer, and is a universal computer, *i.e.*, it is *Turing Complete*, if given access to infinite memory.

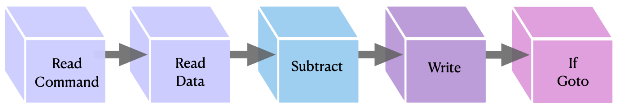


Figure 2. Graphical representation of the building blocks necessary to implement the OISC instruction. The first two blocks transfer the data/command to the scratchpad, the second and third implement the substraction and store the result, while the last one implements the if goto command that completes the instruction.

Algorithm 2 `SUBLEQ`(a, b, c)

```
mem[$b$] = mem[$b$] - mem[$a$] mem[$b$] $\leq 0$ goto instruction $c$ goto next instruction
```

The following describes the construction of a looped transformer that can execute a program written in a specific set of instructions. The transformer keeps track of the lines of code, memory locations, and a program counter, using the memory part of the input as memory registers and the command part as lines of code/instructions. The scratchpad is used to record the additions and pointers involved in each instruction, and the read, write, and conditional branch operations are utilized.

Lemma 1. *There exists a looped transformer architecture that can run SUBLEQ programs. This architecture has nine layers, two heads, and a width of $O(\log(n) + N)$, where n is the length of the input sequence that is proportional to the length of the program and memory used by the emulated OISC, and N is the number of bits we use to store each integer. The integers are considered to be in the range* $[-2^{N-1}+1, 2^{N-1}-1]$

Before we present our construction some observations are in place.

The importance of loops. The use of a loop outside the transformer is crucial as it allows the computer to keep track of the program counter and execute the instructions in the correct order. Without this loop, the size of the transformer would have to scale with the number of lines of code, making the implementation impractical. Note that the overall complexity of running a `SUBLEQ` program is going to scale with the number of lines of code, which is to be expected given standard complexity theoretic assumptions on the circuit depth of functions. Note however that the depth of the looped transfromer itself does not scale with the size of the program.

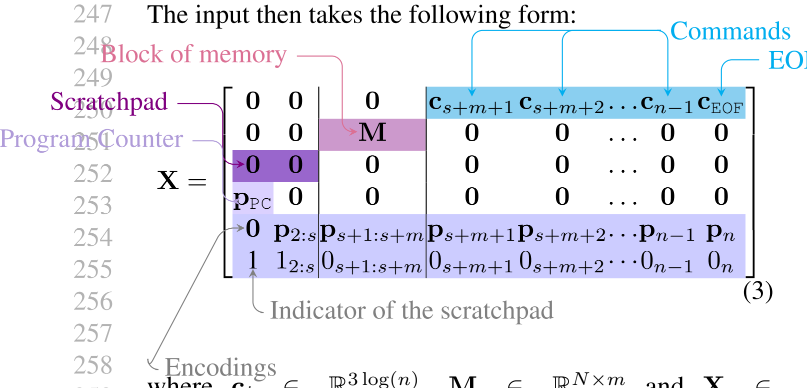
[2] This version of the `SUBLEQ` instruction is a slightly restricted version of the original instruction; here we separate the memory / registers from the instructions. We show that this restriction does not make our version computationally less powerful by proving in **??** that our version is also Turing Complete.

OISC as a basis for a more flexible attention-based computer. The following construction describes an implementation of a fully functioning one-instruction set computer (OISC) using a transformer architecture. The memory stores integers and the instructions are executed in a sequential manner. The key to this construction is the reverse engineering of the attention mechanism to perform read/write operations and taking full advantage of each piece of the transformer architecture, including the feedforward layers. This implementation serves as the foundation for a more general attention-based computer presented in the next subsection, where the subtraction of two contents of memory can be replaced with a general function, allowing for the implementation of arbitrary iterative algorithms.

Proof of Lemma 1. Looking at Alg. 2, note that each instruction can be specified by just 3 indices, a, b, and c. Since we use binary representation of indices to form positional encodings and pointers, each of these indices can be represented by a $\log n$ dimensional vector. We represent each instruction by simply concatenating these embedding vectors to form a $3\log n$ dimensional vector as follows:

$$\mathbf{c} = \begin{bmatrix} \mathbf{p}_a \\ \mathbf{p}_b \\ \mathbf{p}_c \end{bmatrix}.$$

The input then takes the following form:



$$\mathbf{X} = \left[\begin{array}{cc|c|ccccc} \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{c}_{s+m+1} & \mathbf{c}_{s+m+2} & \cdots & \mathbf{c}_{n-1} & \mathbf{c}_{\texttt{EOF}} \\ \mathbf{0} & \mathbf{0} & \mathbf{M} & \mathbf{0} & \mathbf{0} & \dots & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \dots & \mathbf{0} & \mathbf{0} \\ \mathbf{p}_{\texttt{PC}} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \dots & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{p}_{2:s} & \mathbf{p}_{s+1:s+m} & \mathbf{p}_{s+m+1} & \mathbf{p}_{s+m+2} & \cdots & \mathbf{p}_{n-1} & \mathbf{p}_n \\ 1 & 1_{2:s} & 0_{s+1:s+m} & 0_{s+m+1} & 0_{s+m+2} & \cdots & 0_{n-1} & 0_n \end{array}\right] \tag{3}$$

where $\mathbf{c}_i \in \mathbb{R}^{3\log(n)}$, $\mathbf{M} \in \mathbb{R}^{N\times m}$ and $\mathbf{X} \in \mathbb{R}^{(8\log(n)+3N+1)\times n}$. The first s columns constitute the scratchpad, the next m constitute the memory section, and the last $n-m-s$ columns contain the instructions.

The program counter, $\mathbf{p}_{\texttt{PC}}$ points to the next instruction that is to be executed, and hence it is initialized to the first instruction as $\mathbf{p}_{\texttt{PC}} := \mathbf{p}_{s+m+1}$. The contents of the memory section are N dimensional ± 1 binary vectors which represent the corresponding integers. We follow the 2's complement convention to represent the integers, described as follows. Let's say the bits representing an integer are $b_N, \dots, b_1$, with b_N being the most significant bit. Then,

1. If $b_N = -1$, then the integer is considered positive with the value $\sum_{i=1}^{N-1} 2^{i-1}\frac{b_i+1}{2}$.
2. If $b_N = +1$, then the integer is considered negative with the value $-2^{N-1} + \sum_{i=1}^{N-1} 2^{i-1}\frac{b_i+1}{2}$.

Step 1 - Read the instruction $\mathbf{c}_{\texttt{PC}}$. The first thing to do is to read and copy the instruction pointed to by $\mathbf{p}_{\texttt{PC}}$ in the scratchpad. The current instruction is located at column index $\texttt{PC}$, and is pointed to by the current program counter $\mathbf{p}_{\texttt{PC}}$. The instruction, $\mathbf{c}_{\texttt{PC}}$ consists of three pointers, each of length $\log n$. In particular we copy the elements at the location $(1:3\log(n), \texttt{PC})$ to the location $(3\log(n)+4 : 6\log(n)+3, 1)$. This can be done using the `read` operation as described in **??**. Hence, after this operation, the input looks as follows:

$$\mathbf{X} = \left[\begin{array}{cc|c|ccccc} \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{c}_1 & \mathbf{c}_2 & \cdots & \mathbf{c}_{n-m-s} & \mathbf{c}_{\texttt{EOF}} \\ \mathbf{0} & \mathbf{0} & \mathbf{M} & \mathbf{0} & \mathbf{0} & \dots & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \dots & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \dots & \mathbf{0} & \mathbf{0} \\ \mathbf{c}_{\texttt{PC}} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \dots & \mathbf{0} & \mathbf{0} \\ \mathbf{p}_{\texttt{PC}} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \dots & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{p}_{2:s} & \mathbf{p}_{s+1:s+m} & \mathbf{p}_{s+m+1} & \mathbf{p}_{s+m+2} & \cdots & \mathbf{p}_{n-1} & \mathbf{p}_n \\ 1 & 1_{2:s} & 0_{s+1:s+m} & 0_{s+m+1} & 0_{s+m+2} & \cdots & 0_{n-1} & 0_n \end{array}\right]$$

$$= \left[\begin{array}{cc|c|ccccc} \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{c}_1 & \mathbf{c}_2 & \cdots & \mathbf{c}_{n-m-s-1} & \mathbf{c}_{\texttt{EOF}} \\ \mathbf{0} & \mathbf{0} & \mathbf{M} & \mathbf{0} & \mathbf{0} & \dots & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \dots & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \dots & \mathbf{0} & \mathbf{0} \\ \mathbf{p}_a & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \dots & \mathbf{0} & \mathbf{0} \\ \mathbf{p}_b & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \dots & \mathbf{0} & \mathbf{0} \\ \mathbf{p}_c & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \dots & \mathbf{0} & \mathbf{0} \\ \mathbf{p}_{\texttt{PC}} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \dots & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{p}_{2:s} & \mathbf{p}_{s+1:s+m} & \mathbf{p}_{s+m+1} & \mathbf{p}_{s+m+2} & \cdots & \mathbf{p}_{n-1} & \mathbf{p}_n \\ 1 & 1_{2:s} & 0_{s+1:s+m} & 0_{s+m+1} & 0_{s+m+2} & \cdots & 0_{n-1} & 0_n \end{array}\right]$$

This step can be done in one layer.

Step 2 - Read the data required by the instruction. We need to read the data that the columns a, b contain. To do so, we again use the `read` operation on the pointers $\mathbf{p}_a, \mathbf{p}_b$. Note that we need two heads for this operation, one each for reading a and b. The resulting output sequence looks like

$$\mathbf{X} = \left[\begin{array}{cc|c|ccccc} \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{c}_1 & \mathbf{c}_2 & \cdots & \mathbf{c}_{n-m-s-1} & \mathbf{c}_{\texttt{EOF}} \\ \mathbf{0} & \mathbf{0} & \mathbf{M} & \mathbf{0} & \mathbf{0} & \dots & \mathbf{0} & \mathbf{0} \\ \mathrm{mem}[a] & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \dots & \mathbf{0} & \mathbf{0} \\ \mathrm{mem}[b] & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \dots & \mathbf{0} & \mathbf{0} \\ \mathbf{p}_a & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \dots & \mathbf{0} & \mathbf{0} \\ \mathbf{p}_b & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \dots & \mathbf{0} & \mathbf{0} \\ \mathbf{p}_c & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \dots & \mathbf{0} & \mathbf{0} \\ \mathbf{p}_{\texttt{PC}} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \dots & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{p}_{2:s} & \mathbf{p}_{s+1:s+m} & \mathbf{p}_{s+m+1} & \mathbf{p}_{s+m+2} & \cdots & \mathbf{p}_{n-1} & \mathbf{p}_n \\ 1 & 1_{2:s} & 0_{s+1:s+m} & 0_{s+m+1} & 0_{s+m+2} & \cdots & 0_{n-1} & 0_n \end{array}\right]. \tag{4}$$

This step can be done in one layer.

Step 3 - Perform subtraction. Let $\boldsymbol{x}$ denote a column of the input $\mathbf{X}$. Let it have the following structure:

$$\boldsymbol{x} = \begin{bmatrix} * & * & \boldsymbol{b}_r & \boldsymbol{b}_s & * & * & * & * & * \end{bmatrix}^\top,$$

where each entry above represents the corresponding column element of the matrix $\mathbf{X}$ in (4). Thus, $\boldsymbol{b}_r = \texttt{mem}[a], \boldsymbol{b}_s = \texttt{mem}[b]$ for the first column, and $\boldsymbol{b}_r = \boldsymbol{b}_s = \mathbf{0}$ otherwise.

Hence, to perform $\boldsymbol{b}_{s-r}$, we first need to compute the binary representation of $-r$, which is $\boldsymbol{b}_{-r}$, and then simply add it to $\boldsymbol{b}_s$. To compute $\boldsymbol{b}_{-r}$, which is the 2's complement of $\boldsymbol{b}_r$, we just need to flip the bits of $\boldsymbol{b}_r$ and add 1. Bit flipping a ± 1 bit can be done with a neuron simply as $b_{\text{flipped}} = 2 * \sigma(-b) - 1$. For adding 1, we can use **??**. Hence, each of these operations can be done using 1 ReLU layer of width $O(N)$, and so we need 2 transformer layers to perform this (Here we make the intermediate attention layers become the identity mapping by setting their value matrices to $\mathbf{0}$). Finally, we need one more ReLU layer to add $\boldsymbol{b}_s$ to $\boldsymbol{b}_{-r}$, hence bringing the total to 3 transformer layers. This results in calculating $\texttt{mem}[b] - \texttt{mem}[a]$ and place it above $\mathbf{p}_a$.

Step 4 - Write the result back to memory. Writing $\text{mem}[b] - \text{mem}[a]$ back to location b can be done using the pointer $\mathbf{p}_b$ and the set of embeddings and applying the `write` operation described in **??**. This operation requires one layer.

Step 5 - Conditional branching. We first use **??** as described in **??** to create the flag, which is 1 if $\text{mem}[b] - \text{mem}[a] \leq 0$ and 0 otherwise. This can be done using the Equation (1b) of the transformer. Thus, we have

$$\mathbf{X} = \left[\begin{array}{cc|c|ccccc} \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{c}_1 & \mathbf{c}_2 & \ldots & \mathbf{c}_{n-m-s-1} & \mathbf{c}_{\texttt{EOF}} \\ \mathbf{0} & \mathbf{0} & \mathbf{M} & \mathbf{0} & \mathbf{0} & \ldots & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \ldots & \mathbf{0} & \mathbf{0} \\ \text{flag} & 0 & 0 & 0 & 0 & \ldots & 0 & 0 \\ \mathbf{p}_a & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \ldots & \mathbf{0} & \mathbf{0} \\ \mathbf{p}_b & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \ldots & \mathbf{0} & \mathbf{0} \\ \mathbf{p}_c & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \ldots & \mathbf{0} & \mathbf{0} \\ \mathbf{p}_{\texttt{PC}} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \ldots & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{p}_{2:s} & \mathbf{p}_{s+1:s+m} & \mathbf{p}_{s+m+1} & \mathbf{p}_{s+m+2} & \cdots & \mathbf{p}_{n-1} & \mathbf{p}_n \\ 1 & 1_{2:s} & 0_{s+1:s+m} & 0_{s+m+1} & 0_{s+m+2} & \cdots & 0_{n-1} & 0_n \end{array}\right] \tag{5}$$

This operation requires one layer.

Next we use the construction described in **??** to choose, depending on the value of the flag, whether we want to increment the current program counter or we want to jump in the command c. Similar to **??**, this step needs 2 layers of transformers.

Step 6 - Error Correction. Note that some of the steps above we incur some error while reading and writing due to the fact that we are using softmax instead of hardmax. This error can be made arbitrarily small by increasing the temperature of the softmax. In this step, we push the error down to zero. Note that all the elements of $\mathbf{X}$ can only be one of $\{-1, 0, 1\}$, with some additive error from reads and writes as explained before. Assume that the temperature is set high enough that the error is at most $\epsilon < 0.5$. Then, a noisy bit b can be fixed using the following ReLU:

$$\begin{aligned} b_{\text{noiseless}} = & \frac{1}{1-2\epsilon}(\sigma(b+1-\epsilon) - \sigma(b+\epsilon)) \\ & + \frac{1}{1-2\epsilon}(\sigma(b-\epsilon) - \sigma(b-1+\epsilon)) - 1. \end{aligned}$$

This operation can be done with a single layer of transformer.

Step 7 - Program Termination. The special command $\mathbf{c}_{\texttt{EOF}}$ is used to signal the end of a program to the transformer. This command is made up of three encodings: $\mathbf{p}_{s+1}, \mathbf{p}_{s+2}$, and $\mathbf{p}_n$. The first encoding, $\mathbf{p}_{s+1}$, points to the first entry in the memory, which we hard-code to contain the value 0. The second encoding, $\mathbf{p}_{s+2}$, points to the second entry in the memory, which is hard-codeded to contain the value -1. The third encoding, $\mathbf{p}_n$, points to itself, signaling the end of the program and preventing further execution of commands. Hence, on executing this command, the next command pointer is set to point to this command again. This ensures that the transformer maintains the final state of the input.

- For this, we ensure that the last instruction in each program is $\mathbf{c}_{\texttt{EOF}}$, and that $\texttt{mem}[s+1] = 0$ and $\texttt{mem}[s+2] = -1$.
- For this case $a = s+1$, $b = s+2$, and $c = n$.
- The memory is updated with the value $\text{mem}[b] = \text{mem}[b] - \text{mem}[a]$. Since $\text{mem}[a] = 0$ here, the memory remains unchanged.
- Since $\text{mem}[b] \leq 0$ here, the branch is always true and thus the pointer for the next instruction is again set to point to $\mathbf{c}_{\texttt{EOF}}$.

$\square$

4.2 `FLEQ`: A More Flexible Attention-based Computer

In this section, we introduce `FLEQ`, a generalization of `SUBLEQ` that defines a more flexible reduced-instruction set computer. This implied set of additional instructions is based on a more advanced version of `SUBLEQ` that allows for the implementation of multiple functions within the same transformer network. This is achieved by generalizing the previous OISC construction to include not just addition of registers, but any function from a set of M predefined functions implementable by a transformer network. In the following, we use the term `FLEQ` to refer interchangably to the instruction, the language, and the attention-based computer it defines.

The design of `FLEQ` allows for the implementation of complex and sophisticated algorithms by generating more general functions beyond simple subtraction, such as matrix multiplication, computation of square roots, activation functions, etc. This not only increases the flexibility of the system, but also makes it possible to implement nonlinear computations, linear algebra calculations, and iterative optimization algorithms for in-context learning while containing the length of the corresponding programs.

Definition 1. *Let $\mathcal{T}_i$ be a transformer network of the form* (1) *with l_i-layers, h_i-heads and dimensionality r. We call this a **"transformer-based function block"** if it implements a function $f(\mathbf{A}, \mathbf{B})$ where the input and output sequence format is assumed to be the following: $\mathbf{A} \in \mathbb{R}^{d_h \times d_w}$ is assumed to be provided in the first set of d columns (columns 1 to d) and $\mathbf{B} \in \mathbb{R}^{d_h \times d_w}$ the second set of d columns (columns $d+1$ to $2d$); after passing the input through the l_i layers, the output of $f(\mathbf{A}, \mathbf{B}) \in \mathbb{R}^{d_h \times d_w}$ is stored in the third d columns (columns $2d+1$ to $3d$), where d is the maximum size that the input could have and it is a constant that we determine. Note that $d_h, d_w \leq d$. Finally, the sequence length of the block is $s \geq 3d$. Similarly to d, s is a predetermined constant.*

The parameters $\mathbf{A}, \mathbf{B}$ can be scalars, vectors or matrices as long as they can fit within a $d \times d$ matrix. Hence, the above definition is minimally restrictive, with the only main constraint being the input and output locations. More details about the input and output requirements will be explained towards the end of this subsection.

Theorem 1. *Given M different transformer-based function blocks $\mathcal{T}_1, \cdots, \mathcal{T}_M$, there exists a transformer $\mathcal{T}$ of the form* (1) *with number of layers $9 + \max\{l_1, \cdots, l_M\}$, a number of $\sum_{i=1}^{M} h_i$ heads , and dimensionality $O(Md + \log n)$ such that running it recurrently T times can run T instructions of any program where each instruction is $\texttt{FLEQ}(a, b, c, m, \text{flag}, p, d_h, d_w)$, and executes the following:*

$$\texttt{mem}[c] = f_m(\texttt{mem}[a], \texttt{mem}[b]) \quad ; \quad \text{if } \texttt{mem}[\text{flag}] \leq 0 \text{ goto instruction } p \tag{6}$$

Here n is the total length of the program and we assume that $\texttt{mem}[flag]$ is an integer. The parameters d_h, d_w are explained in Remark 1 below.

Remark 1. *Note that, the transformer $\mathcal{T}$ contains M transformer-based function blocks and each one may use different input parameters. We thus define with d the max length that each of the parameters $\mathbf{A}, \mathbf{B}, \mathbf{C}$ (stored in locations a, b, c) as in Definition 1 can have; this is a global constant and it is fixed for all the different instances that we can create. Now, d_h, d_w refer to the maximum dimension that the parameters can have in a specific instance of the transformer $\mathcal{T}$; the rest of the columns $d - d_w$ and rows $d - d_h$ are set to zero.*

The proof of this theorem can be found in **??**. Below we explain some of our design choices.

Execution cycle of the unified attention-based computer. In each iteration of the looped transformer, one instruction is fetched from the set of instructions in the input according to the program counter. The instruction is then copied to the scratchpad. Depending on the function to be implemented, a different function block location is used to locally record the results of that function. Once the result is calculated, it is copied back to a specified memory location provided by the instruction. The execution cycle is similar to the one-instruction set computer (OISC) in the previous section, with the main difference being that for each instruction, we can choose from a pre-selected list of functions that take inputs in the form of arbitrary arrays of numbers, such as matrices, vectors, and scalars.

The format of the input sequence. In Fig. 3, we illustrate the input $\mathbf{X}$ to our looped transformer, which can execute a program written as a series of FLEQ instructions. Note that $\mathbf{X}$ is divided into three sections: Scratchpad, Memory, and Instructions. As in the left bottom part of Fig. 3, we allocate a separate part of the scratchpad for each of the M functions that are internally implemented by the transformer. For example, if we have matrix multiplication and element-wise square root as two functions, we would allocate a different function block for each one.

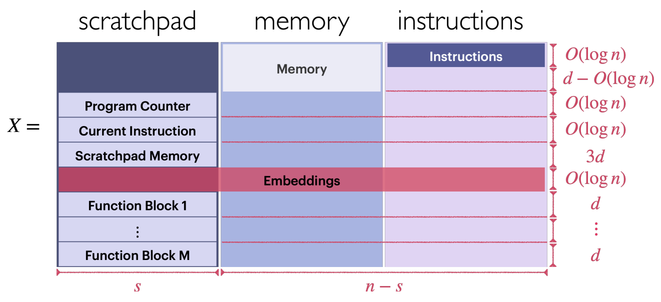
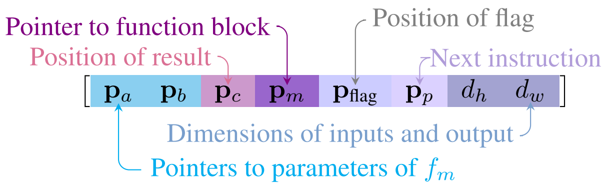


Figure 3. The structure of input $\mathbf{X}$, to execute FLEQ commands.

This design may not be the most efficient, but our goal is to demonstrate the possibilities of looped transformers. Additionally, since the number of different functions is typically small in the applications we have in mind, the design does not significantly increase in size. The choice to reserve different function blocks for each predefined function is for convenience, as it allows for separate treatment of functions without worrying about potentially overlapping results. We believe that a design with a single function block is feasible, but it would significantly complicate the rest of the transformer construction.

Instruction format. The instruction in Theorem 1 is essentially a composition of the following two components: the function call to f_m and the conditional branching (if ... goto ...). The instruction, located at the top right side of Fig. 3 contains the following components:


Pointer to function block
Position of result
Position of flag
Next instruction
$\mathbf{p}_a$ $\mathbf{p}_b$ $\mathbf{p}_c$ $\mathbf{p}_m$ $\mathbf{p}_{\text{flag}}$ $\mathbf{p}_p$ d_h d_w
Dimensions of inputs and output
Pointers to parameters of f_m
(7)

The goal of each positional encoding vector in Equation (7) is to point to the corresponding space of the input where each component required by the instruction is located. To be specific, $\mathbf{p}_a$ and $\mathbf{p}_b$ point to the locations that the inputs a and b are located, $\mathbf{p}_c$ points to the location to which we will record the final result of the function f_m. Similarly, $\mathbf{p}_m$ points to the function block in the scratchpad that the intermediate computations required for f_m are recording, $\mathbf{p}_{\text{flag}}$ points to the variable that we check if it is non-positive (the result is used for conditional branching), and $\mathbf{p}_p$ points to the address of the line of code that we would jump if the variable in pointed by $\mathbf{p}_{\text{flag}}$ is non-positive.

Execute a function; Jump to command. Recall that the first four parameters (a, b, c, m) of FLEQ, as well as the last two (d_h, d_w) are related to the implementation of the function block, while the other two (flag, p) are related with the conditional branching. Since there is no overlap between the two components of each instruction, it is possible to use each of these components independently. By having a fixed location flag_0 where $\texttt{mem}[\text{flag}_0]$ is always set to 1, we can have the simpler command $\texttt{FLEQ}(a, b, c, m, \text{flag}_0, p, d_h, d_w)$ which implements

$$\texttt{mem}[c] = f_m(\texttt{mem}[a], \texttt{mem}[b]).$$

Further, by having fixed locations a_0, b_0, c_0 which are not used elsewhere in the program, and hence inconsequential, we can have the simpler command $\texttt{FLEQ}(a_0, b_0, c_0, m, \text{flag}, p, d_h, d_w)$ which implements

$$\text{if } \texttt{mem}[\text{flag}] \leq 0 \text{ goto instruction } p.$$

Using this, we get the following corollary:

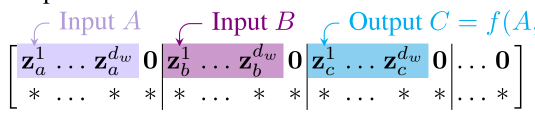
Corollary 1. *The Unified Attention Based Computer presented in Theorem 1 can run programs where each instruction can be **either** of the following two simple instructions:*

- $\texttt{mem}[c] = f_m(\texttt{mem}[a], \texttt{mem}[b])$
- if $\texttt{mem}[\text{flag}] \leq 0$ goto instruction p

Format of Transformer-Based Function Blocks. Recall that each function block is located at the bottom left part of the input $\mathbf{X}$, as shown in Fig. 3. Each transformer-based function block is expected to operate using the following format of the input:

- The number of rows in the input is r, while the number of columns is s and $s \geq 3d$. Here s will dictate the total maximum number of columns that any transformer-based function block needs to operate. The reason that s might be larger than $3d$ has to do with the fact that some blocks may need some extra scratchpad space to perform some calculations.
- The function block specifies the dimensions of input and output. Say they are $d_h \times d_w$, where $d_h, d_w \leq d$. These will be part of the instruction which calls this function inside the FLEQ framework, as in (7).
- Suppose each function block has two inputs ($\mathbf{A} \in \mathbb{R}^{d_h \times d_w}$ and $\mathbf{B} \in \mathbb{R}^{d_h \times d_w}$) and one output $f(\mathbf{A}, \mathbf{B}) = \mathbf{C} \in \mathbb{R}^{d_h \times d_w}$. As in (8), the function block is divided into four parts: (1) the first input $\mathbf{A}$ is placed in the first d_h rows and the first d_w columns, (2) the second input $\mathbf{B}$ is placed in the first d_h rows and the columns $d + 1 : d + d_w$, (3) the output $f(\mathbf{A}, \mathbf{B}) = \mathbf{C}$ is in the first d_h rows and the columns $2d+1 : 2d+d_w$ columns and 4) the rest $s - 3d$ column used as scratchpad space for performing necessary calculations. Note that the unused columns are set to zero.
- The last $r - d_h$ rows can be used by the transformer-based function block in any way, *e.g.*, to store any additional positional encodings.

We put the format of the input of each *transformer-based function block* in (8). The first input $\mathbf{A} = [\boldsymbol{z}_a^1, \cdots, \boldsymbol{z}_a^{d_w}]$ of the function is zero padded and stored in the first d columns. Similarly, the second input $\mathbf{B} = [\boldsymbol{z}_b^1, \cdots, \boldsymbol{z}_b^{d_w}]$ is stored in the next d columns. The output/result of the function block $\mathbf{C} = [\boldsymbol{z}_c^1, \cdots, \boldsymbol{z}_c^{d_w}]$ is located in the next d columns while we have some extra $s - 3d$ columns which can be used as scratchpad.


Input A
Input B
Output $C = f(A, B)$
$\mathbf{z}_a^1 \dots \mathbf{z}_a^{d_w}$ $\mathbf{0}$ $\mathbf{z}_b^1 \dots \mathbf{z}_b^{d_w}$ $\mathbf{0}$ $\mathbf{z}_c^1 \dots \mathbf{z}_c^{d_w}$ $\mathbf{0}$ $\dots$ $\mathbf{0}$
$* \dots * *$ $* \dots * *$ $* \dots * *$ $\dots *$
(8)

5 Applications

Our unified template allows us to implement algorithms and iterative operations as programs. Calculations like multiplication, division, square root, etc., as well as linear algebra functions like matrix multiplication, transposition can be formed as attention-based function blocks. One key component of our analysis for creating non-linear functions is the manipulation of the softmax in Equation (1a) so as to create the sigmoid function $g(x) = 1/(1 + e^{-x})$. We then encode a different sigmoid function at each head and create linear combinations of them to create approximations for different functions. For more details see **??**.

Using these function-blocks and the FLEQ transformer, we are further able to implement a calculator, inversion, power iteration and learning algorithms like SGD on a linear model with square loss, as well as, full backpropagation on a 2-layer sigmoid-activated neural network. We now formally state some of these results below, for a complete list, please see the appendix.

Calculator. Our first result is the emulation of a simple calculator. To prove the Lemma below, we use **??**, which provides error guarantees in terms of the number of heads m, to approximate the square root and the inversion function. The details can be found in **??**.

Lemma 2. *There exists a transformer with* 12 *layers,* m *heads and dimensionality* $O(\log n)$ *that uses the Unified Attention Based Computer framework in Section 4.2 to implement a calculator which can perform addition, subtraction, multiplication, and computing the inverse, square root and percentage. For computing the inverse and square root, the operand needs to be in the range* $[-e^{O(m)}, -\tilde{\Omega}(\frac{1}{\sqrt{m}})] \cup [\tilde{\Omega}(\frac{1}{\sqrt{m}}), e^{O(m)}]$ *and* $[0, O(m^2)]$ *respectively, and the returned output is correct up to an error of* $O(1/\sqrt{m})$ *and* $O(1/m)$ *respectively. Here,* n *is the number of operations to be performed.*

Linear Algebra. We continue with emulating approximation algorithms like the Newton-Raphson Method to find the inverse of a non-singular matrix $\mathbf{A}$ (Alg. 3), and the Power Iteration Algorithm for finding the eigenvector corresponding to the eigenvalue with the maximum absolute value (Alg. 4). Notice that once we have established matrix transposition, matrix multiplication and functions like scalar division etc., these algorithms can be encoded as sequential applications of those results.

Algorithm 3 Pseudocode for Matrix Inversion .

```
1: $\mathbf{X}_{-T} = \epsilon\mathbf{A}$
2: for $i = -T, \ldots, 0$ do
3:     $\mathbf{X}_{i+1} = \mathbf{X}_i(2\mathbf{I} - \mathbf{A}\mathbf{X}_i)$
4: end for
```

Lemma 3. *Consider a matrix* $\mathbf{A} \in \mathbb{R}^{d\times d}$*, then for any* $\epsilon > 0$ *there exists a transformer with 13 layers, 1 head and dimensionality* $r = O(d)$ *that emulates Alg. 3 with output* $\mathbf{X}_1^{(transf)}$ *that satisfies* $\|\mathbf{X}_1^{(transf)} - \mathbf{X}_1\| \le \epsilon$*. This error* ϵ *arises due to softmax, and can be driven arbitrarily close to 0 by increasing the temperature.*

Algorithm 4 Power Iteration

```
Input: $\mathbf{A}, T$
1: Initialize $b_0 = \mathbf{1}$
2: for $k = 0, \ldots, T-1$ do
3:     $\mathbf{b}_{k+1} = \mathbf{A}\mathbf{b}_k$
4: end for
5: $\mathbf{b} = \mathbf{b}_T/\|\mathbf{b}_T\|$
```

Lemma 4. *Consider a matrix* $\mathbf{A} \in \mathbb{R}^{d\times d}$*, then for any* $\epsilon > 0$ *there exists a transformer with* 13 *layers, 1 head and dimensionality* $r = O(d)$ *that emulates Alg. 4 for* $T = O(\log 1/\epsilon)$ *iterations with output* $\mathbf{b}_{T+1}^{(transf)}$ *that satisfies* $\|\mathbf{b}_{T+1}^{(transf)} - \mathbf{b}_{T+1}\| \le \epsilon$*. This error* ϵ *arises due to softmax, and can be driven arbitrarily close to 0 by increasing the temperature.*

Stochastic Gradient Descent and Backpropagation. Finally, we present our result on the emulation of stochastic gradient descent (SGD) in 2-layer neural networks, over a set of data points $(\mathbf{x}_i, y_i)$. We first implement Alg. 5, which serves as a function for calculating and updating the weight and bias matrices with steps proportional to their gradients. Each function call takes as input pointers to the weight and biases matrices, one data point and its corresponding label and the step-size .

Algorithm 5 Backpropagation

```
Define: Loss function: $J(x) = \frac{1}{2}x^2$.
Input: $\mathbf{W}_1 \in \mathbb{R}^{m\times d}$, $\mathbf{b}_1 \in \mathbb{R}^m$, $\mathbf{W}_2 \in \mathbb{R}^{m\times 1}$, $\mathbf{b}_2 \in \mathbb{R}$ ,
       $\mathbf{x} \in \mathbb{R}^d$, $y \in \mathbb{R}$, $\eta \in \mathbb{R}$
1: Compute $\boldsymbol{z} = \mathbf{W}_1\mathbf{x} + \mathbf{b}_1$, $\boldsymbol{a} = \sigma(\boldsymbol{z})$.
2: Compute $o = \mathbf{W}_2\boldsymbol{a} + \mathbf{b}_2$.
3: Compute $\delta_2 = (o - y)$.
4: Compute $\delta_1 = \sigma'(\boldsymbol{z}) \odot \mathbf{W}_2(o - y)$.
5: Compute $\frac{\partial J}{\partial \mathbf{W}_2} = \delta_2\boldsymbol{a}^\top$, $\frac{\partial J}{\partial \mathbf{b}_2} = \delta_2$.
6: Compute $\frac{\partial J}{\partial \mathbf{W}_1} = \delta_1\mathbf{x}^\top$, $\frac{\partial J}{\partial \mathbf{b}_1} = \delta_1$.
7: Update $\mathbf{W}_1, \mathbf{W}_2, \delta_1, \delta_2$ with one gradient update.
```

Lemma 5. *There exists a transformer with 13 layers, 1 head and dimensionality* $O(\log(|\mathcal{D}|) + d)$ *that uses the Unified Attention Based Computer framework to implement* T *iterations of SGD on a two-layer sigmoid-activated neural network, over a set of* n_d *data points* $(\mathbf{x}_i, y_i) \in \mathbb{R}^{d+1}$*,* $i = 1, \ldots, |\mathcal{D}|$*. The step size is given as a parameter to the program. The emulation of each step of SGD is not exact, there is some error in each step which, however, can be driven down arbitrarily close to 0 by increasing the temperature of softmax and another free parameter which does not affect the size of the network.*

6 Conclusion

In this work, we have shown that transformer networks can be used as universal computers by programming them with specific weights and placing them in a loop.

References

Akyürek, E., Schuurmans, D., Andreas, J., Ma, T., and Zhou, D. What learning algorithm is in-context learning? investigations with linear models. *arXiv preprint arXiv:2211.15661*, 2022.

Brown, T., Mann, B., Ryder, N., Subbiah, M., Kaplan, J. D., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., et al. Language models are few-shot learners. *Advances in neural information processing systems*, 33: 1877–1901, 2020.

Chowdhery, A., Narang, S., Devlin, J., Bosma, M., Mishra, G., Roberts, A., Barham, P., Chung, H. W., Sutton, C., Gehrmann, S., et al. Palm: Scaling language modeling with pathways. 2022.

Chung, H. W., Hou, L., Longpre, S., Zoph, B., Tay, Y., Fedus, W., Li, E., Wang, X., Dehghani, M., Brahma, S., et al. Scaling instruction-finetuned language models. *arXiv preprint arXiv:2210.11416*, 2022.

Dasgupta, I., Lampinen, A. K., Chan, S. C., Creswell, A., Kumaran, D., McClelland, J. L., and Hill, F. Language models show human-like content effects on reasoning. *arXiv preprint arXiv:2207.07051*, 2022.

Dehghani, M., Gouws, S., Vinyals, O., Uszkoreit, J., and Kaiser, Ł. Universal transformers. *arXiv preprint arXiv:1807.03819*, 2018.

Dosovitskiy, A., Beyer, L., Kolesnikov, A., Weissenborn, D., Zhai, X., Unterthiner, T., Dehghani, M., Minderer, M., Heigold, G., Gelly, S., et al. An image is worth 16x16 words: Transformers for image recognition at scale. In *International Conference on Learning Representations*, 2020.

Garg, S., Tsipras, D., Liang, P., and Valiant, G. What can transformers learn in-context? a case study of simple function classes. In *Advances in Neural Information Processing Systems*, 2022.

Hutchins, D., Schlag, I., Wu, Y., Dyer, E., and Neyshabur, B. Block-recurrent transformers. *arXiv preprint arXiv:2203.07852*, 2022.

Kenton, J. D. M.-W. C. and Toutanova, L. K. Bert: Pre-training of deep bidirectional transformers for language understanding. In *Proceedings of NAACL-HLT*, pp. 4171–4186, 2019.

Khan, S., Naseer, M., Hayat, M., Zamir, S. W., Khan, F. S., and Shah, M. Transformers in vision: A survey. *ACM computing surveys (CSUR)*, 54(10s):1–41, 2022.

Kirsch, L. and Schmidhuber, J. Meta learning backpropagation and improving it. In Beygelzimer, A., Dauphin, Y., Liang, P., and Vaughan, J. W. (eds.), *Advances in Neural Information Processing Systems*, 2021. URL https://openreview.net/forum?id=hhU9TEvB6AF.

Lewkowycz, A., Andreassen, A., Dohan, D., Dyer, E., Michalewski, H., Ramasesh, V., Slone, A., Anil, C., Schlag, I., Gutman-Solo, T., et al. Solving quantitative reasoning problems with language models. *arXiv preprint arXiv:2206.14858*, 2022.

Lindner, D., Kramár, J., Rahtz, M., McGrath, T., and Mikulik, V. Tracr: Compiled transformers as a laboratory for interpretability. *arXiv preprint arXiv:2301.05062*, 2023.

Liu, B., Ash, J. T., Goel, S., Krishnamurthy, A., and Zhang, C. Transformers learn shortcuts to automata. *arXiv preprint arXiv:2210.10749*, 2022.

Mavaddat, F. and Parhami, B. Urisc: the ultimate reduced instruction set computer. *International Journal of Electrical Engineering Education*, 25(4):327–334, 1988.

Merrill, W., Sabharwal, A., and Smith, N. A. Saturated transformers are constant-depth threshold circuits. *Transactions of the Association for Computational Linguistics*, 10:843–856, 2022.

Nye, M., Andreassen, A. J., Gur-Ari, G., Michalewski, H., Austin, J., Bieber, D., Dohan, D., Lewkowycz, A., Bosma, M., Luan, D., et al. Show your work: Scratchpads for intermediate computation with language models. 2021.

Pérez, J., Barceló, P., and Marinkovic, J. Attention is turing-complete. *Journal of Machine Learning Research*, 22(75): 1–35, 2021. URL http://jmlr.org/papers/v22/20-302.html.

Pérez, J., Marinković, J., and Barceló, P. On the turing completeness of modern neural network architectures, 2019. URL https://arxiv.org/abs/1901.03429.

Shen, Z., Liu, Z., and Xing, E. Sliced recursive transformer. In *European Conference on Computer Vision*, pp. 727–744. Springer, 2022.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, Ł., and Polosukhin, I. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

von Oswald, J., Niklasson, E., Randazzo, E., Sacramento, J., Mordvintsev, A., Zhmoginov, A., and Vladymyrov, M. Transformers learn in-context by gradient descent. *arXiv preprint arXiv:2212.07677*, 2022.

Wei, C., Chen, Y., and Ma, T. Statistically meaningful approximation: a case study on approximating turing machines with transformers. *Advances on Neural Information Processing Systems (NeurIPS)*, 2022a.

Wei, J., Tay, Y., Bommasani, R., Raffel, C., Zoph, B., Borgeaud, S., Yogatama, D., Bosma, M., Zhou, D., Metzler, D., et al. Emergent abilities of large language models. *arXiv preprint arXiv:2206.07682*, 2022b.

Wei, J., Wang, X., Schuurmans, D., Bosma, M., Chi, E., Le, Q., and Zhou, D. Chain of thought prompting elicits reasoning in large language models. *arXiv preprint arXiv:2201.11903*, 2022c.

Weiss, G., Goldberg, Y., and Yahav, E. Thinking like transformers. In *International Conference on Machine Learning*, pp. 11080–11090. PMLR, 2021.

Yuan, L., Chen, Y., Wang, T., Yu, W., Shi, Y., Jiang, Z.-H., Tay, F. E., Feng, J., and Yan, S. Tokens-to-token vit: Training vision transformers from scratch on imagenet. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pp. 558–567, 2021.

Yun, C., Bhojanapalli, S., Rawat, A. S., Reddi, S., and Kumar, S. Are transformers universal approximators of sequence-to-sequence functions? In *International Conference on Learning Representations*, 2019.

Zhou, H., Nova, A., Larochelle, H., Courville, A., Neyshabur, B., and Sedghi, H. Teaching algorithmic reasoning via in-context learning. *arXiv preprint arXiv:2211.09066*, 2022.