LOOPED TRANSFORMERS AS PROGRAMMABLE COMPUTERS

Anonymous authors
Paper under double-blind review

ABSTRACT

We present a framework for using transformer networks as universal computers by programming them with specific 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.

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). 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 multidigit 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 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; Esolangs).

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

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

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.

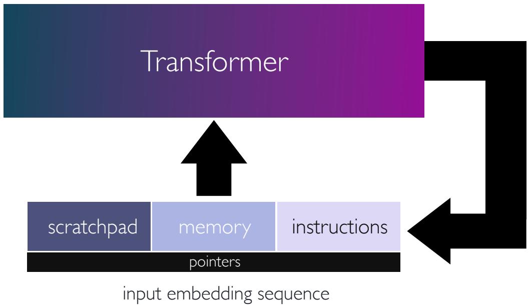

Figure 1: A sketch of the looped transformer architecture, where the input sequence stores the commands, memory where the data is read/written from, and a scratchpad where intermediate results are stored. The input is processed by the network and the output is used as the new input, allowing the network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

2 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; the superscript i that appears on the weight matrices indicates those corresponding to the i-th attention head.Consistent with previous literature, the first equation Equation (1a) represents the attention layer. We refer to the combination of attention and ReLU layers as a single transformer layer.

Iterative computation through a simple 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*.

Feeding the output back to its input 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.

To be able to build compute boxes out of a TF network, it is crucial to format the input sequence $\mathbf{X}$ in a way that separates memory, a cache-like scratchpad, and commands.

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 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. This is the central location where the inputs and outputs of all computation are recorded.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. Regardless of the specific computation that is performed, 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. When computation is needed, the data is first copied from the memory to the scratchpad, where it is updated and transformed as necessary. Once the computation is complete, the updated data is then returned and copied back to the memory for future use or reference.

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.

3 Main Results

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 instantiations of this single instruction. A variant of such an instruction is SUBLEQ, where different registers, or memory locations are accessed. The way that SUBLEQ works is simple as shown in Alg. 1. 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.

Algorithm 1 SUBLEQ(a, b, c)

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

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 ten layers, two heads, and a width of $O(\log(n) + log(N))$, 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* $[-2^{N-1}+1, 2^{N-1}-1]$.

FLEQ: A More Flexible Attention-based Computer. We introduce FLEQ, a generalization of SUBLEQ that defines a more flexible reduced-instruction set computer, which includes 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.

Theorem 1 (Informal). *Given M different functions $\{f_m\}_{m=1}^{M}$, that each needs at most L layers of transformer to be implemented, there exists a transformer with $9+L$ layers such that running it recurrently T times can run T instructions of any program where each instruction is* FLEQ$(a, b, c, m, \text{flag},)$, *and executes the following:*

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

Applications. Our unified template, introduced above, allows us to implement algorithms and iterative operations as programs, consisting of FLEQ instructions.

The first key component is that calculations like multiplication, division, square root, etc., as well as linear algebra functions like matrix multiplication, transposition can be formed with at most 4 layers of the transformer architecture, in a template form which we call attention-based function blocks. This specific form allows to use these blocks in a plug-and-play form in our unified template. These function blocks are: addition, division, square root and generally all functions that can be approximated by a sum of sigmoids, any type of matrix,vector, scalar multiplication, transpose of a matrix and read/write operations.

As a result, 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.* For the formal definitions, theorems, proofs and a complete list of our results, please see the appendix.

4 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.

Barron, A. Universal approximation bounds for superpositions of a sigmoidal function. *IEEE Transactions on Information Theory*, 39(3):930–945, 1993. doi: 10.1109/18.256500.

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.

Charton, F. Linear algebra with transformers. *arXiv preprint arXiv:2112.01898*, 2021.

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.

Esolangs. Subleq. URL `https://esolangs.org/wiki/Subleq`.

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.

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.

Perekrestenko, D., Grohs, P., Elbrächter, D., and Bölcskei, H. The universal approximation power of finite-width deep relu networks. *arXiv preprint arXiv:1806.01528*, 2018.

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.

CONTENTS

A 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. A useful element of their analysis is the fact that language is completely removed from the picture, and they perform all operations on the level of vector embeddings. This allows a higher abstraction level than using language as an input, and in fact is what also allows us to obtain our derivations.

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.

In a similar vein, von Oswald et al. (2022) argues that training Transformers on auto-regressive tasks is closely related to gradient-based meta-learning formulations. The authors also provide a hard-coded 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).

B Building Transformer Blocks towards General Computation

To build general compute boxes using transformer networks, specialized compute blocks are required. These blocks will be assembled to create the desired end functionality. In this section, we highlight various operations that transformer layers can perform. These operations will serve as building blocks to create more complex routines and algorithms.

B.1 Positional Encodings, Program Counter, and Data Pointers

To aid the transformer in locating the position of each token, each column of $\mathbf{X}$ is appended with positional encodings that is based on the column index. In this case, similar to Wei et al. (2022a), the positional encodings is the binary representation of the column index, to keep the encoding dimension low, *i.e.*, logarithmic in the sequence length. This approach to using positional encodings is slightly different from the typical method of adding them to the embeddings of the input sequence. However, in this case, appending them as suffixes to the embeddings allows for cleaner arguments and constructions.

In particular, the encoding for token/column indexed by i is a $\log(n)$-dimensional ± 1 binary vector $\mathbf{p}_i \in \{\pm 1\}^{\log(n)}$, where n is the length of the input sequence. Using the standard binary representation of an integer i, meaning $i = \sum_{k=0}^{\log(n)-1} 2^k \cdot b_k$, the positional encoding vector $\mathbf{p}_i$ is set to -1 at index j if the binary representation of i has 0 at the j-th index, *i.e.*, $b_i = 0$, otherwise it is $+1$. As a result, we have $\mathbf{p}_i^T \mathbf{p}_i = \log(n)$ and by Cauchy-Schwarz inequality, $\mathbf{p}_i^T \mathbf{p}_j < |\mathbf{p}_i||\mathbf{p}_j| = \sqrt{\log(n)}\sqrt{\log(n)} = \log(n)$ whenever $i \neq j$, since $\mathbf{p}_i, \mathbf{p}_j$ differ in at least one coordinate.

In the applications presented, the transformer often needs to execute iterative algorithms or go through a sequence of commands. To achieve this, we utilize a program counter that iterates through the commands. The counter contains the encoding of the location where the next command is stored. Additionally, a command may have data pointers that point to the location of the data the command needs to read and write to. Both the program counter and data pointers utilize the same positional encodings as discussed in the previous paragraph. Using binary vectors as positional encodings allows us to easily increment the program counter by 1 (or any other amount) using the feed forward ReLU layers in the transformer architecture (1). This is formalized in the following lemma, for the proof see Lemma 10.

Lemma 2. *Given two d-dimensional binary vectors representing two non-negative integers, there exists a 1-hidden layer feedforward network with ReLU activation, containing $8d$ activations in the hidden layer and d neurons in the output layer, that can output the binary vector representation of their sum, as long as the sum is less than 2^{d+1}.*

Furthermore, this technique for pointing to specific data locations enables the transformer to effectively read and write from/to data during the execution of the algorithm or sequence of commands that is build to implement.

B.2 read / write: Copying Data/Instructions to/from the Scratchpad

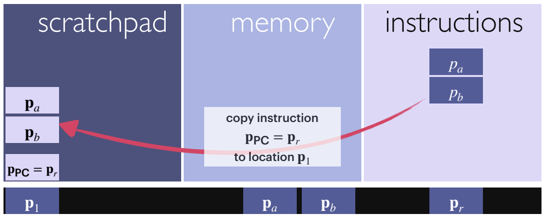

Figure 2: A sketch of the read operation. Arrows show command blocks being copied from the part of the input that is allocated to commands to the scratchpad. Typically an instruction is another set of pointers. Positional encodings and counters are used for tracking what is copied where.

As previously stated, the scratchpad serves as a temporary memory for storing all information needed for computation. This includes copying commands and data to it, performing computation, and writing results back to memory. This process has similarities with the copy mechanism developed in Akyürek et al. (2022).

The following lemma states that the command or data pointed to by the program counter or a data pointer in the current command can be copied to the scratchpad. The location of the program counter is conventionally placed right below the contents of the scratchpad, but it can be changed arbitrarily. Keeping it in a specific location throughout the entire computation helps retain a good organization of the construction.

Lemma 3 (`read`). *A transformer with one layer, one head, and width of $O(\log n + d)$, where d is the dimension of the data vectors and n is the length of the input, can read data/command vectors from the input to the scratchpad from the location pointed to by the position embedding vector in the scratchpad. This operation incurs an error which can be driven arbitrarily close to 0 by increasing the temperature of the softmax operation.*

The next lemma explains that a vector stored in the scratchpad can be copied to a designated location in memory, as specified within the scratchpad itself. This allows for the transfer of data from the scratchpad to a specific location in memory for further use or storage.

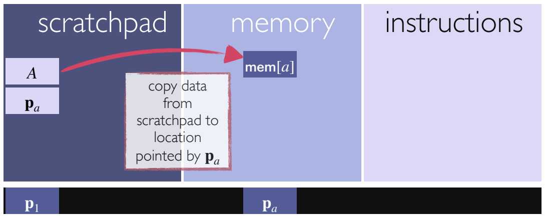

Figure 3: A sketch of the write operation. Arrows show data blocks being copied from the scratchpad to a designated location in the part of the input allocated for memory. Positional encodings are used for tracking the destination location and ensuring data is written at the correct memory location.

Lemma 4 (`write`). *A transformer network with a single layer, one head, and width $O(\log n + d)$, where d is the dimension of the data vectors and n is the length of the input, can effectively write a data vector stored in the scratchpad to a specific location in the input, as designated by a positional encoding vector in the scratchpad. This operation incurs an error which can be driven arbitrarily close to 0 by increasing the temperature of the softmax operation.*

B.3 IF $\langle$*condition*$\rangle$ THEN GOTO $\langle$*instruction*$\rangle$

In this subsection, we state the main ideas used to implement a conditional branching instruction that evaluates a condition and sets the program counter to a specified location if the condition is true, or increments the program counter by 1 if the condition is false. The form of the command is as follows: `if` mem$[a] \leq 0$, `then goto` i, where mem$[a]$ is a value of some location in the memory part of the input sequence. This command has two parts: evaluating the inequality and modifying the program counter accordingly.

The first thing we do is read from `mem`$[a]$, as described in the previous subsection. We then use one ReLU layer to create the "flag", the condition. This is implemented for the cases that `mem`$[a]$ contains an integer, or its binary representation as in Appendix C.1.

Let the current Program Counter be $\mathbf{p}_{\text{PC}}$, which points to a given command. Thus, if flag is 1, we want the program counter to "jump" and become $\mathbf{p}_i$, else if flag is 0 the program counter will be incremented by one, and set to be $\mathbf{p}_{\text{PC}+1}$. This can be implemented with one ReLU layer, which selects one of the vectors based on the value of the flag. The details of this construction are given in Appendix F.3.

C EMULATING A SINGLE INSTRUCTION COMPUTER

C.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 instantiations of this single instruction. 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 to the next instruction from the current line of code.[1] 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.

Algorithm 2 SUBLEQ(a, b, c)

```
mem[$b$] = mem[$b$] - mem[$a$]
**if** mem[$b$] $\leq 0$ **then**
    goto instruction $c$
**else**
    goto next instruction
**end if**
```

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.

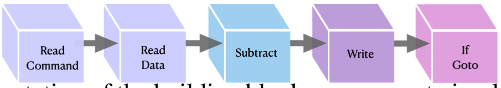

Figure 4: Graphical representation of the building blocks necessary to implement the OISC instruction. The first two blocks transfer the data/command to the scratchpad, the second and third implement the subtraction and store the result, while the last one implements the if goto command that completes the instruction.

Lemma 5. *There exists a looped transformer architecture that can run SUBLEQ programs. This architecture has ten layers, two heads, and a width of $O(\log(n) + log(N))$, 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* $\left[-2^{N-1}+1, 2^{N-1}-1\right]$.

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.

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.

We defer the proof of Lemma 5 in Appendix G, which provides the details of constructing a looped transformer architecture to run SUBLEQ programs.

[1] 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 Appendix H that our version is also Turing Complete.

C.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, which includes 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 easier implementation of complex algorithms using functions beyond simple subtraction, such as matrix multiplication, computation of square roots, activation functions, etc. directly as built-in primitives in the architecture. 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 are explained in Appendix I.

Theorem 2. *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:*

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

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. The execution of this operation incurs an error which can be driven arbitrarily close to 0 by increasing the temperature of the softmax operation.

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 Appendix K. 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. For a more detailed overview of FLEQ and how it interacts with the Transformer-based Function Blocks, we refer the reader to Appendix I.

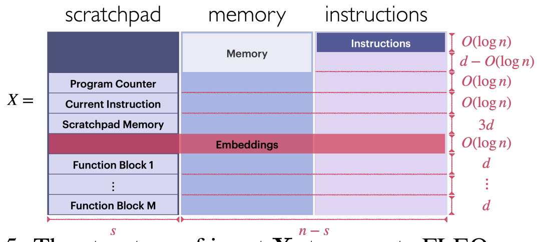

Figure 5: The structure of input $\mathbf{X}$, to execute FLEQ commands.

Computational concerns: Do we need full attention? In our constructions we can reduce the computational complexity of the attention mechanism by limiting it the number of embedding vectors that each part of the input has to attend to. In our specific construction, only the columns within the scratchpad require global attention. By focusing only on these columns, we can reduce the computational complexity of the attention mechanism from $O(n^2 d)$ to $O(nd)$, where n is the number of input sequences, d is the dimension of the embedding vectors.

D 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 Lemma 11.

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 Lemma 11, 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 Appendix M.

Lemma 6. *There exists a transformer with* 12 *layers,* m *heads and dimensionality* $O(\log n)$ *that uses the Unified Attention Based Computer framework in Section C.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 .

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

Lemma 7. *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*

$\|\mathbf{X}_1^{(transf)} - \mathbf{X}_1\| \leq \epsilon$. *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 $k = 0, \dots, T-1$ do
3:     $\mathbf{b}_{k+1} = \mathbf{A}\mathbf{b}_k$
4: end for
5: $\mathbf{b} = \mathbf{b}_T / \|\mathbf{b}_T\|$
```

Lemma 8. *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 $\|\mathbf{b}_{T+1}^{(transf)} - \mathbf{b}_{T+1}\| \leq \epsilon$. 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 9. *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 $(\mathbf{x}_i, y_i) \in \mathbb{R}^{d+1}$, $i = 1, \dots, |\mathcal{D}|$. 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.*

E LIMITATIONS

In this paper, we have presented a new approach for using transformer blocks for function approximation. However, there are several limitations to our work. However, there are several limitations to our work that should be considered. One limitation is that the constructions presented in this paper have not been experimentally validated for efficiency. Additionally, implementing a transformer-based approach may be less efficient than running the algorithm directly. Furthermore, our constructions are forced to have a specific input structure where commands and memory are separated, which may lead to inefficiencies. At this point, it is also unclear how to combine hardcoded transformer models with pretrained ones. Lastly, we have not conducted a thorough finite precision analysis of our algorithms. Despite these limitations, our work presents a novel approach to exploring the mechanics of attention based networks and our methods may have potential for further exploration and development.

F OMITTED PROOFS

F.1 ADDITION OF POINTERS.

Lemma 10. *There exists a 1-hidden layer feedforward, ReLU network, with $6d$ activations in the hidden layer and d neurons in the output layer that when given two d-dimensional binary vectors representing two non-negative integers, can output the binary vector representation of their sum, as long as the sum is less than 2^{d+1}.*

Proof. For the purpose of explaining this proof, we use the $\{0,1\}^d$ binary representation of the integers, instead of the $\{\pm 1\}^d$ binary representation. However, since the conversion of a bit between the two representations can be done easily using simple affine transformation, the proof will also work for the $\{\pm 1\}^d$ binary representation.

Let the two integers be a, b and let $c := a + b$. We assume that $c < 2^d$. Futher, let a_1 be the least significant bit of a, a_d the most significant, and a_i be the i-th most significant bit, and similarly for b and c. Further, let $a_{[i]}$ represent the integer formed by considering only the least i significant bits of a.

Note that c_i is only dependent on the least i bits of a and b, and not on the more significant bits of a or b. In particular, c_i only depends on $a_{[i]} + b_{[i]}$. Define $s := a_{[i]} + b_{[i]}$, and note that $c_i = s_i$. Further note that $s < 2^{i+1}$ and hence can be represented in $i+1$ bits. Then, whenever $c_i = 1$, there can be two cases: $(s_{i+1} = 1, s_i = 1)$; or $(s_{i+1} = 0, s_i = 1)$. This can be equivalently written as $c_i = 1$ iff $s \in [2^{i-1}, 2^i - 1] \cup [3 \cdot 2^{i-1}, 2^{i+1} - 1]$. This can be computed by the following ReLU:

$$\begin{aligned} c_i = {} & (\sigma(s - 2^{i-1} + 1) - \sigma(s - 2^{i-1})) + (\sigma(2^i - s) - \sigma(2^i - s - 1)) - 1 \\ & + (\sigma(s - 3 \cdot 2^{i-1} + 1) - \sigma(s - 3 \cdot 2^{i-1})). \end{aligned}$$

Thus, each bit of c can be computed using 6 neurons. Hence, computing the entire sum needs $8d$ activations. □

F.2 READ/WRITE OPERATIONS.

Lemma 3 (`read`)**.** *A transformer with one layer, one head, and width of $O(\log n + d)$, where d is the dimension of the data vectors and n is the length of the input, can read data/command vectors from the input to the scratchpad from the location pointed to by the position embedding vector in the scratchpad. This operation incurs an error which can be driven arbitrarily close to 0 by increasing the temperature of the softmax operation.*

Proof. Consider a simplified input where the scratchpad only has one column, and we have positional encodings, denoted as $\mathbf{p}_i$, that point to the location where data or commands should be copied from. In this case, the operation we want to perform is as follows:

$$\left[\begin{array}{c|ccccc} \mathbf{0} & \boldsymbol{v}_2 & \cdots & \boldsymbol{v}_i & \cdots \\ \boldsymbol{v}_1 & \mathbf{0} & \cdots & \mathbf{0} & \cdots \\ \mathbf{p}_i & \mathbf{0} & \cdots & \mathbf{0} & \cdots \\ \mathbf{0} & \mathbf{p}_2 & \cdots & \mathbf{p}_i & \cdots \\ \mathbf{0} & \mathbf{0} & \cdots & \mathbf{0} & \cdots \\ 1 & 0 & \dots & 0 & \dots \end{array}\right] \rightarrow \left[\begin{array}{c|ccccc} \mathbf{0} & \boldsymbol{v}_2 & \cdots & \boldsymbol{v}_i & \cdots \\ \boldsymbol{v}_i & \mathbf{0} & \cdots & \mathbf{0} & \cdots \\ \mathbf{p}_i & \mathbf{0} & \cdots & \mathbf{0} & \cdots \\ \mathbf{0} & \mathbf{p}_2 & \cdots & \mathbf{p}_i & \cdots \\ \mathbf{0} & \mathbf{0} & \cdots & \mathbf{0} & \cdots \\ 1 & 0 & \dots & 0 & \dots \end{array}\right]$$

which moves data/command embedding vector $\boldsymbol{v}_i$ from the memory/command part of the input to the scratchpad. The first row contains the data to be read, the second row has the data written in the scratchpad, the third row contains the program counter, the fourth row contains the positional encodings, the fifth row is used by for temporary storage and the last row is just a bit that indicates whether the column is in the scratchpad or not.

We use the following key and query matrices: $\mathbf{K} = \mathbf{Q} = [\mathbf{0} \quad \mathbf{0} \quad \mathbf{I} \quad \mathbf{I} \quad \mathbf{0} \quad 0]$, so that the key and query become equal to $\mathbf{KX} = \mathbf{QX} = [\ \mathbf{p}_i \quad \mathbf{p}_2 \quad \cdots \quad \mathbf{p}_i \quad \cdots\ ]$, and hence,

$$(\mathbf{KX})^\top \mathbf{QX} = \begin{bmatrix} \mathbf{p}_i^\top \mathbf{p}_i & \mathbf{p}_i^\top \mathbf{p}_2 & \cdots \\ \mathbf{p}_2^\top \mathbf{p}_i & \mathbf{p}_2^\top \mathbf{p}_2 & \cdots \\ \vdots & \vdots & \vdots \\ \mathbf{p}_i^\top \mathbf{p}_i & \mathbf{p}_i^\top \mathbf{p}_2 & \cdots \\ \vdots & \vdots & \vdots \end{bmatrix}$$

Recall that $\mathbf{p}_i$ is a $\log(n)$-dimensional ± 1 vector such that $\mathbf{p}_i^T \mathbf{p}_i = \log(n)$ and each $\mathbf{p}_i^T \mathbf{p}_j \leq \log(n) - 1$ for $j \neq i$. We show in the appendix that if we apply the softmax with temperature $\lambda \geq \log \frac{n^3}{\epsilon}$, we have $\sigma_{\mathrm{S}}((\mathbf{KX})^\top \mathbf{QX})$ to be an $n \times n$ matrix of the following form

$$\begin{bmatrix} \frac{1}{2} & 0 & 0 & \cdots & \frac{1}{2} & \cdots & 0 \\ 0 & 1 & 0 & \cdots & 0 & \cdots & 0 \\ 0 & 0 & 1 & \cdots & 0 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \ddots & \vdots \\ \frac{1}{2} & 0 & 0 & \cdots & \frac{1}{2} & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & 0 & \cdots & 1 \end{bmatrix} + \epsilon \mathbf{M} = \begin{bmatrix} \frac{\boldsymbol{e}_1 + \boldsymbol{e}_i}{2} & \boldsymbol{e}_2 & \boldsymbol{e}_3 & \cdots & \frac{\boldsymbol{e}_1 + \boldsymbol{e}_i}{2} & \cdots \end{bmatrix} + \epsilon \mathbf{M},$$

where $\boldsymbol{e}_i$ is the ith column of the identity matrix, $\|\mathbf{M}\| \leq 1$, and ϵ is as defined in Appendix L. For the purpose of the proof, we ignore the error term $\epsilon \mathbf{M}$, because it can be reduced arbitrarily by increasing the temperature (it can be made precisely equal to 0, if we consider hardmax instead of softmax), and overall does not limit us from deriving arbitrarily small error bounds.

Next we set the output and value weight matrices as follows

$$\mathbf{V} = \begin{bmatrix} \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & 0 \\ \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & 0 \\ \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & 0 \\ \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & 0 \\ \mathbf{I} & \mathbf{I} & \mathbf{0} & \mathbf{0} & \mathbf{0} & 0 \\ \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & 0 \end{bmatrix}.$$

Using this, the output of the head is

$$\mathbf{X} + \mathbf{VX}\sigma_{\mathrm{S}}((\mathbf{KX})^\top \mathbf{QX}) = \left[\begin{array}{c|cccc} \mathbf{0} & \boldsymbol{v}_2 & \cdots & \boldsymbol{v}_i & \cdots \\ \boldsymbol{v}_1 & \mathbf{0} & \cdots & \mathbf{0} & \cdots \\ \mathbf{p}_i & \mathbf{0} & \cdots & \mathbf{0} & \cdots \\ \mathbf{0} & \mathbf{p}_2 & \cdots & \mathbf{p}_i & \cdots \\ \frac{\boldsymbol{v}_1 + \boldsymbol{v}_i}{2} & \boldsymbol{v}_2 & \cdots & \frac{\boldsymbol{v}_1 + \boldsymbol{v}_i}{2} & \cdots \\ 1 & 0 & \cdots & 0 & \cdots \end{array}\right]$$

Each column above has the following form:

$$\begin{bmatrix} \boldsymbol{v}_{\text{orig}}^{0} \\ \boldsymbol{v}_{\text{orig}}^{1} \\ \boldsymbol{v}_{\text{orig}} \\ \mathbf{p}^{(0)} \\ \mathbf{p}^{(1)} \\ \boldsymbol{v}_{\text{new}} \\ b \end{bmatrix},$$

where $\boldsymbol{v}_{\text{orig}}^{(0)}$ and $\boldsymbol{v}_{\text{orig}}^{(1)}$ are the original value vectors (present in the top two row blocks) contained in that column, $\mathbf{p}^{(0)}$ and $\mathbf{p}^{(1)}$ are the corresponding embeddings of each column, $\boldsymbol{v}_{\text{new}}$ is the new value, and b is the bit indicating whether the column is part of the scratchpad or not.

The feedforward layers have the following form:

$$\begin{aligned}
&\boldsymbol{v}^{(1)}_{\text{orig}} := \boldsymbol{v}^{(1)}_{\text{orig}} + \sigma(C(b-1)\mathbf{1} + 2\boldsymbol{v}_{\text{new}} - 2\boldsymbol{v}^{(1)}_{\text{orig}}) - \sigma(C(b-1)\mathbf{1} - 2\boldsymbol{v}_{\text{new}} + 2\boldsymbol{v}^{(1)}_{\text{orig}}) \\
&\boldsymbol{v}_{\text{new}} := \boldsymbol{v}_{\text{new}} - \sigma(\boldsymbol{v}_{\text{new}}) + \sigma(-\boldsymbol{v}_{\text{new}}) = \mathbf{0},
\end{aligned}$$

where C is a large positive constant. The first equation is performing the operation of subtracting $\boldsymbol{v}_{\text{new}}$ from $\boldsymbol{v}_{\text{orig}}$ but only when the sum and difference of $C(b-1)\mathbf{1}$ and $\boldsymbol{v}_{\text{new}}$ are positive, otherwise the subtraction does not occur. The second equation is resetting the value of $\boldsymbol{v}_{\text{new}}$ to zero after it has been copied to $\boldsymbol{v}_{\text{orig}}$, where $\sigma(-\boldsymbol{v}_{\text{new}})$ is the rectified linear unit (ReLU) applied to the negative of $\boldsymbol{v}_{\text{new}}$.

It can be verified that the output of the feedforward layers would then be the desired result

$$\mathbf{X} = \left[\begin{array}{c|ccccc}
\mathbf{0} & \boldsymbol{v}_2 & \cdots & \boldsymbol{v}_i & \cdots \\
\boldsymbol{v}_i & \mathbf{0} & \cdots & \mathbf{0} & \cdots \\
\mathbf{p}_i & \mathbf{0} & \cdots & \mathbf{0} & \cdots \\
\mathbf{0} & \mathbf{p}_2 & \cdots & \mathbf{p}_i & \cdots \\
\mathbf{0} & \mathbf{0} & \cdots & \mathbf{0} & \cdots \\
1 & 0 & \ldots & 0 & \ldots
\end{array}\right].$$

□

Lemma 4 (`write`). *A transformer network with a single layer, one head, and width $O(\log n + d)$, where d is the dimension of the data vectors and n is the length of the input, can effectively write a data vector stored in the scratchpad to a specific location in the input, as designated by a positional encoding vector in the scratchpad. This operation incurs an error which can be driven arbitrarily close to 0 by increasing the temperature of the softmax operation.*

Proof. We want to achieve the following operation

$$\mathbf{X} = \left[\begin{array}{c|ccccc}
\mathbf{0} & \boldsymbol{v}_2 & \cdots & \boldsymbol{v}_i & \cdots \\
\boldsymbol{v}_1 & \mathbf{0} & \cdots & \mathbf{0} & \cdots \\
\mathbf{p}_i & \mathbf{0} & \cdots & \mathbf{0} & \cdots \\
\mathbf{0} & \mathbf{p}_2 & \cdots & \mathbf{p}_i & \cdots \\
\mathbf{0} & \mathbf{0} & \cdots & \mathbf{0} & \cdots \\
1 & 0 & \ldots & 0 & \ldots
\end{array}\right] \rightarrow \left[\begin{array}{c|ccccc}
\mathbf{0} & \boldsymbol{v}_2 & \cdots & \boldsymbol{v}_1 & \cdots \\
\boldsymbol{v}_1 & \mathbf{0} & \cdots & \mathbf{0} & \cdots \\
\mathbf{p}_i & \mathbf{0} & \cdots & \mathbf{0} & \cdots \\
\mathbf{0} & \mathbf{p}_2 & \cdots & \mathbf{p}_i & \cdots \\
\mathbf{0} & \mathbf{0} & \cdots & \mathbf{0} & \cdots \\
1 & 0 & \ldots & 0 & \ldots
\end{array}\right],$$

The construction for this is identical to the one for `read` (see the proof of Lemma 3), except that the feedforward layers are outputting the following:

$$\begin{aligned}
&\boldsymbol{v}^{(0)}_{\text{orig}} := \boldsymbol{v}^{(0)}_{\text{orig}} + \sigma(-Cb\mathbf{1} + 2\boldsymbol{v}_{\text{new}} - 2\boldsymbol{v}^{(0)}_{\text{orig}}) + \sigma(-Cb\mathbf{1} - 2\boldsymbol{v}_{\text{new}} + 2\boldsymbol{v}^{(0)}_{\text{orig}}) \\
&\boldsymbol{v}_{\text{new}} := \boldsymbol{v}_{\text{new}} - \sigma(\boldsymbol{v}_{\text{new}}) + \sigma(-\boldsymbol{v}_{\text{new}}) = \mathbf{0},
\end{aligned}$$

where C is a large positive constant. The first equation updates the value of a vector $\boldsymbol{v}_{\text{orig}}$ in memory with the value of a vector $\boldsymbol{v}_{\text{new}}$ from the scratchpad. The second equation is resetting the new vector in the scratchpad to zero. It can be verified that the output of the feedforward layers would be

$$\mathbf{X} = \left[\begin{array}{c|ccccc}
\mathbf{0} & \boldsymbol{v}_2 & \cdots & \boldsymbol{v}_1 & \cdots \\
\boldsymbol{v}_1 & \mathbf{0} & \cdots & \mathbf{0} & \cdots \\
\mathbf{p}_i & \mathbf{0} & \cdots & \mathbf{0} & \cdots \\
\mathbf{0} & \mathbf{p}_2 & \cdots & \mathbf{p}_i & \cdots \\
\mathbf{0} & \mathbf{0} & \cdots & \mathbf{0} & \cdots \\
1 & 0 & \ldots & 0 & \ldots
\end{array}\right].$$

□

F.3 IF $\langle$*condition*$\rangle$ THEN GOTO $\langle$*instruction*$\rangle$: CONDITIONAL BRANCHING

In this subsection, we will implement a conditional branching instruction that evaluates a condition and sets the program counter to a specified location if the condition is true, or increments the program counter by 1 if the condition is false. The form of the command is as follows: `if` $\text{mem}[a] \leq 0$, `then goto` i, where $\text{mem}[a]$ is a value of some location in the memory part of the input sequence. This command has two parts: evaluating the inequality and modifying the program counter accordingly.

The first thing we do is read from mem$[a]$, as described in the previous subsection. Let us say that "flag" is the truth value of the inequality. Since we assume that for such conditional branching command, mem$[a]$ contains an integer, the following ReLU network can be used to compute the flag:

$$\text{flag} = 1 - \sigma(\texttt{mem}[a]) + \sigma(\texttt{mem}[a] - 1). \tag{3}$$

In Appendix C.1, we consider $\texttt{mem}[a]$ to be vectors contain the binary ± 1 representation of integers. There we use 2's complement convention to represent negative integers. Let the vector be $[b_N \ \dots \ b_1]$, where b_N is the most significant bit and b_1 the least significant. As we explain in that section, the sign of b_N indicates whether the integer is negative or positive (The number is negative if $b_N = +1$ and non-negative otherwise). Hence, the flag is 1 if $b_N = +1$ or if all the bits are -1 (which is the case when $\texttt{mem}[a]$ represents the integer 0).

$$\text{flag} = \sigma(b_N) + \sigma\left(1 + N - \sum_{i=1}^{N} b_i\right). \tag{4}$$

Let the current Program Counter be $\mathbf{p}_{\texttt{PC}}$, which points to a given command. Thus, if flag is 1, we want the program counter to "jump" and become $\mathbf{p}_i$, else if flag is 0 the program counter will be incremented by one, and set to be $\mathbf{p}_{\texttt{PC}+1}$.

Consider that the simplified input currently has the following scratchpad

$$\begin{bmatrix} * & * & \dots & * & * \\ \text{flag} & \mathbf{0} & \dots & \mathbf{0} & \mathbf{0} \\ \mathbf{p}_{\texttt{PC}} & \mathbf{0} & \dots & \mathbf{0} & \mathbf{0} \\ \mathbf{p}_i & \mathbf{0} & \dots & \mathbf{0} & \mathbf{0} \end{bmatrix},$$

where $'*'$ are inconsequential values. The incremented pointer, $\mathbf{p}_{\texttt{PC}+1}$, can be computed using the pointer incrementing operation that we described in the Subsection B.1, using one feedforward layer of (1b).Then,

$$\begin{aligned} \mathbf{p}_{\text{next}} = {} & 2\sigma(\mathbf{p}_{\texttt{PC}+1} - \mathbf{1}\text{flag}) \\ & + 2\sigma(\mathbf{p}_i - \mathbf{1}(1 - \text{flag})) - 1, \end{aligned}$$

where $\mathbf{1}$ is the all ones vector. Notice that we can implement this with just the feed forward layers of Equation (1b). To account for the residual connection we can add the expression $-\sigma(\mathbf{p}_{\texttt{PC}}) + \sigma(-\mathbf{p}_{\texttt{PC}})$ in the equation above.

Hence, this entire operation requires 3 feed forward layers of Equation (1b), and hence 2 transformer layers. Note that to ensure that the attention layer of the transformer do not modify the input, we simply set the $\mathbf{V}$ matrix to zero in (1a).

G SUBLEQ: Proof of Lemma 5

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:

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

where $\mathbf{c}_i \in \mathbb{R}^{3\log(n)}$, $\mathbf{M} \in \mathbb{R}^{1\times m}$ and $\mathbf{X} \in \mathbb{R}^{(8\log(n)+2\log(N)+1)\times n}$. 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 Appendix B.2. Hence, after this operation, the input looks as follows:

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

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

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

$$
\mathbf{X} = \left[\begin{array}{cc|c|ccccc}
\mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{c}_1 & \mathbf{c}_2 & \dots & \mathbf{c}_{n-m-s-1} & \mathbf{c}_{\texttt{EOF}} \\
\mathbf{0} & \mathbf{0} & \mathbf{M} & \mathbf{0} & \mathbf{0} & \dots & \mathbf{0} & \mathbf{0} \\
\mathrm{mem}[a] & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \dots & \mathbf{0} & \mathbf{0} \\
\mathrm{mem}[b] & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \dots & \mathbf{0} & \mathbf{0} \\
\mathbf{p}_a & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \dots & \mathbf{0} & \mathbf{0} \\
\mathbf{p}_b & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \dots & \mathbf{0} & \mathbf{0} \\
\mathbf{p}_c & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \dots & \mathbf{0} & \mathbf{0} \\
\mathbf{p}_{\texttt{PC}} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \dots & \mathbf{0} & \mathbf{0} \\
\mathbf{0} & \mathbf{p}_{2:s} & \mathbf{p}_{s+1:s+m} & \mathbf{p}_{s+m+1} & \mathbf{p}_{s+m+2} & \dots & \mathbf{p}_{n-1} & \mathbf{p}_n \\
1 & 1_{2:s} & 0_{s+1:s+m} & 0_{s+m+1} & 0_{s+m+2} & \dots & 0_{n-1} & 0_n
\end{array}\right]. \tag{6}
$$

Commands

EOF

Block of memory

Scratchpad

Program Counter

Encodings

This step can be done in one layer.

Indicator of the scratchpad

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

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

where each entry above represents the corresponding column element of the matrix $\mathbf{X}$ in (6). Thus, $\boldsymbol{b}_r = \texttt{mem}[a], \boldsymbol{b}_s = \texttt{mem}[b]$ for the first column, and $r = s = 0$ 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 Lemma 10. Hence, each of these operations can be done using 1 ReLU layer of width $O(\log N)$, and hence 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 the following:

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

Note that since this can be done in the feedforward layers of the previous step, this does not require an additional layer.

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 Appendix B.2. This operation requires one layer.

Step 5 - Conditional branching. We first use Equation (4) as described in Appendix B.3 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

$$\mathbf{X} = \left[\begin{array}{cc|c|ccccc}
\mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{c}_1 & \mathbf{c}_2 & \dots & \mathbf{c}_{n-m-s-1} & \mathbf{c}_{\texttt{EOF}} \\
\mathbf{0} & \mathbf{0} & \mathbf{M} & \mathbf{0} & \mathbf{0} & \dots & \mathbf{0} & \mathbf{0} \\
0 & 0 & 0 & 0 & 0 & \dots & 0 & 0 \\
\text{flag} & 0 & 0 & 0 & 0 & \dots & 0 & 0 \\
\mathbf{p}_a & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \dots & \mathbf{0} & \mathbf{0} \\
\mathbf{p}_b & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \dots & \mathbf{0} & \mathbf{0} \\
\mathbf{p}_c & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \dots & \mathbf{0} & \mathbf{0} \\
\mathbf{p}_{\texttt{PC}} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \dots & \mathbf{0} & \mathbf{0} \\
\mathbf{0} & \mathbf{p}_{2:s} & \mathbf{p}_{s+1:s+m} & \mathbf{p}_{s+m+1} & \mathbf{p}_{s+m+2} & \cdots & \mathbf{p}_{n-1} & \mathbf{p}_n \\
1 & 1_{2:s} & 0_{s+1:s+m} & 0_{s+m+1} & 0_{s+m+2} & \cdots & 0_{n-1} & 0_n
\end{array}\right] \tag{7}$$

This operation requires one layer.

Next we use the construction described in Appendix B.3 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 Appendix B.3, this step needs 3 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 $\epsilon < 0.5$. 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 $\mathbf{c}_{\texttt{EOF}}$.

H SUBLEQ IS TURING COMPLETE

In this section, we show that our slightly restricted version of the original SUBLEQ instruction (Mavaddat & Parhami, 1988) is indeed also Turing complete. To do this, we will utilize Minsky machines, which are also Turing complete. A Minksy machine comprises of registers and a list of instructions, where each instruction can be either of the following two instructions

- add(a): $\texttt{mem}[a] := \texttt{mem}[a] + 1$, go to the next instruction.
- sub(a, n): If $\texttt{mem}[a] == 0$, go to instruction n. Otherwise $\texttt{mem}[a] := \texttt{mem}[a] - 1$, go to the next instruction.

Given a program written in a language above, we translate it into an equivalent one written in our SUBLEQ language. For this, we initialize three fixed locations / registers c_{-1}, c_0, and c_{+1} such that $\texttt{mem}[c_{-1}] := -1$, $\texttt{mem}[c_0] := 0$, and $\texttt{mem}[c_{+1}] := +1$; as well as an extra register $\texttt{mem}[b]$. We translate the program instruction-by-instruction. Assume that we have translated the first $i-1$ instructions. Let $j-1$ be the index of the last (translated) SUBLEQ instruction, that is, the index of the next SUBLEQ instruction will be j. Then, for the i-th instruction in the Minsky machine language, we translate it into our language as follows:

- Case 1, The i-th instruction of the Minsky machine program is $\texttt{add}(a)$. This is equivalent to $\texttt{SUBLEQ}(a, c_{-1}, j+1)$, and hence the j instruction in our program will simply be $\texttt{SUBLEQ}(a, c_{-1}, j+1)$.
- Case 2, The i-th instruction in the Minsky machine program is $\texttt{sub}(a, n)$. This would be equivalent to the sequence of the following 5 SUBLEQ instructions.

Algorithm 6 Translation for $\texttt{sub}(a, n)$

```
Instr. $j$     : $\texttt{SUBLEQ}(b, b, j+1)$
Instr. $j+1$: $\texttt{SUBLEQ}(b, a, j+3)$
Instr. $j+2$: $\texttt{SUBLEQ}(a, c_{+1}, j+5)$
Instr. $j+3$: $\texttt{SUBLEQ}(a, c_0, n')$
Instr. $j+4$: $\texttt{SUBLEQ}(a, c_{+1}, j+5)$
```

Here n' is the index of the translation of the n-th instruction of the Minsky machine program. This can be computed as a function of the number of `add` and `sub` instructions up to instruction n. The correctness of the above can be verified by considering the three cases: $\texttt{mem}[a] \geq 1$, $\texttt{mem}[a] \leq -1$, and $\texttt{mem}[a] = 0$.

I FLEQ Overview

The format of the input sequence. In Fig. 6, 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. 6, 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.

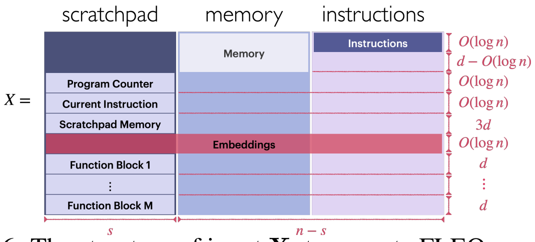

Figure 6: The structure of input $\mathbf{X}$, to execute FLEQ commands.

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 2 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. 6 contains the following components:

$$\begin{array}{r} \\ \text{Pointers to parameters of } f_m \rightarrow \\ \text{Position to write result} \rightarrow \\ \text{Pointer to function block} \rightarrow \\ \text{Position of flag} \rightarrow \\ \text{Next instruction} \rightarrow \\ \\ \text{Dimensions of the inputs and the output} \rightarrow \end{array} \begin{bmatrix} \mathbf{p}_a \\ \mathbf{p}_b \\ \mathbf{p}_c \\ \mathbf{p}_m \\ \mathbf{p}_{\text{flag}} \\ \mathbf{p}_p \\ d_h \\ d_w \end{bmatrix} \tag{8}$$

The goal of each positional encoding vector in Equation (8) 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 2 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. 6. 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 (8).
- 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 (9), 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 (9). 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.

$$\begin{array}{cccc|cccc|cccc|cc}
 & & \text{Input } A & & & & \text{Input } B & & & & \text{Output } C = f(A,B) & & & \\
\mathbf{z}_a^1 & \ldots & \mathbf{z}_a^{d_w} & \mathbf{0} & \mathbf{z}_b^1 & \ldots & \mathbf{z}_b^{d_w} & \mathbf{0} & \mathbf{z}_c^1 & \ldots & \mathbf{z}_c^{d_w} & \mathbf{0} & \ldots & \mathbf{0} \\
* & \ldots & * & * & * & \ldots & * & * & * & \ldots & * & * & \ldots & *
\end{array} \tag{9}$$

Let us consider the case where we wish to multiply a matrix $\mathbf{A} \in \mathbb{R}^{d \times d}$,with a vector $\mathbf{b} \in \mathbb{R}^{d \times 1}$. The resulting output matrix would look as follows:

$$\left[\begin{array}{c|cc|cc|c} \mathbf{A} & \mathbf{b} & \mathbf{0} & \mathbf{A}^\top \mathbf{b} & \mathbf{0} & \mathbf{0} \end{array}\right].$$

J Functions in the Unified Template Form

In this section, we demonstrate how to implement a variety of nonlinear functions and basic linear algebra operations using transformers. These techniques will be crucial in the construction of iterative algorithms in the following sections. Each transformer-based function block in this section fits in our unified template in terms of input/output parameters' locations. We note here that each transformer-based function block might have its own positional encodings used to transfer the output in the correct place or perform some `read/write` operations and they are part of the design of the block.

J.1 Encoding Non-linear Functions within the Attention Mechanism

One key ingredient of our constructions is encoding various functions within the attention mechanism. We do this by forcing the softmax to act as a sigmoid function and by storing multiple coefficients in the query and value weight matrices. As far as we know, this is the first work that shows how general non-linear functions can be emulated by attention layers. This allows us to create linear combinations of sigmoids that can be accessed by an indicator vector in the input. Our analysis is based on the result of Barron (1993) which we present below.

Definition 2. *Let $\Gamma_{C,B}$ be the set of functions defined in a bounded domain B, $f : B \to \mathbb{R}, B \subseteq \mathbb{R}^d$ with a proper extension to $\mathbb{R}^d$ such that they have C bounded Fourier integral,* i.e., *$\int \sup_{x\in B} |w \cdot x|\, F(dw) \leq C$ holds where $F(dw)$ is the magnitude of the Fourier distribution.*

Definition 3. *Given $\tau > 0, C > 0$ and a bounded set B, let*

$$G_{\phi,\tau} = \{\gamma\phi(\tau(\mathbf{a}^T\mathbf{x} + b)) : |\gamma| \leq 2C, \|\mathbf{a}\|_B \leq 1, |b| \leq 1\}$$

where $\|\mathbf{a}\|_B = \sup_{\mathbf{x}\in B}\{\mathbf{x}^T\mathbf{a}\}$ and ϕ is the sigmoid function, i.e., *$\phi(x) = \frac{1}{1+e^{-x}}$.*

Theorem 3 (Theorem 3 in Barron (1993))**.** *Every function $f \in \Gamma_{C,B}$ with $f(0) = 0$ and can be approximated by a linear combination of sigmoids $f_i \in G_{\phi,\tau}$, $i = 1, \ldots m$. If $\tau \geq m^{1/2}\ln m$ the error scales as*

$$\left| f(\mathbf{x}) - \sum_{i=1}^{m} f_i(\mathbf{x}) \right| \leq O\left(\frac{1}{m^{1/2}}\right),\ \mathbf{x} \in B$$

To encode N different functions, we use the index $j \in [N]$ and write $c_{ji}, \mathbf{a}_{ji}$ for the coefficients of the sigmoids that approximate them or

$$f_j(\mathbf{x}) = \sum_{i=1}^{m} c_{ji}\phi(\mathbf{x}^T\mathbf{a}_{ji}) \text{ for } j = 1, \ldots, N$$

We here note that the terms τ, b can be incorporated in the term $\mathbf{a}_{ij}$ by adding an extra coefficient of 1 in $\mathbf{x}$ and multiplying everything with τ.

We are now able to present the lemma on approximating functions using transformer blocks, in a format that is consistent with the FLEQ design outlined in the previous section.

Lemma 11. *Consider an input of the form*

$$\mathbf{X} = \begin{bmatrix} \boldsymbol{e} & \mathbf{0} & \mathbf{x} & \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{p}_{2d+1} & \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \mathbf{p}_1 & \mathbf{p}_{2:d} & \mathbf{0} & \mathbf{p}_{d+2:2d} & \mathbf{p}_{2d+1} & \mathbf{p}_{2d+2:3d} \\ 0 & 0_{2:d} & 1 & 0_{d+2:2d} & 0 & 0_{2d+2:3d} \end{bmatrix} \in \mathbb{R}^{N+d_x\times 3d}.$$

where d is chosen, N is the number of functions we encode and d_x is the dimension of $\mathbf{x}$. $\boldsymbol{e} = \mathbf{e}_j$ an indicator vector of the function we want to choose. Then there exists a transformer-based function block with 3 layers, m heads and dimensionality $O(d)$ such that

$$f(\mathbf{X}) = \begin{bmatrix} * & * & * & * & \sum_{i=1}^{m} c_{ji}\phi(\mathbf{x}^T\mathbf{a}_{ji}) & * \\ \mathbf{0} & \mathbf{0} & \mathbf{x} & \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{p}_{2d+1} & \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \mathbf{p}_1 & \mathbf{p}_{2:d} & \mathbf{0} & \mathbf{p}_{d+2:2d} & \mathbf{p}_{2d+1} & \mathbf{p}_{2d+2:3d} \\ 0 & 0_{2:d} & 1 & 0_{d+2:2d} & 0 & 0_{2d+2:3d} \end{bmatrix}$$

where $*$ denoted inconsequential values that will be ignored downstream.

Proof. The first thing we do is to move the $\mathbf{x}$ to the second row block, as follows:

$$\mathbf{X} = \begin{bmatrix} e & \mathbf{0} & \mathbf{x} & \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{p}_{2d+1} & \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \mathbf{p}_1 & \mathbf{p}_{2:d} & \mathbf{0} & \mathbf{p}_{d+2:2d} & \mathbf{p}_{2d+1} & \mathbf{p}_{2d+2:3d} \\ 0 & 0_{2:d} & 1 & 0_{d+2:2d} & 0 & 0_{2d+2:3d} \end{bmatrix} \to \begin{bmatrix} e & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{x} & \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{p}_{2d+1} & \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \mathbf{p}_1 & \mathbf{p}_{2:d} & \mathbf{0} & \mathbf{p}_{d+2:2d} & \mathbf{p}_{2d+1} & \mathbf{p}_{2d+2:3d} \\ 0 & 0_{2:d} & 1 & 0_{d+2:2d} & 0 & 0_{2d+2:3d} \end{bmatrix}$$

This can be done using a ReLU feedforward layer that performs this using the last row of the input as the indicator bit for the column containing $\mathbf{x}$.

Then we want to create the following transformation

$$\begin{bmatrix} e & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{x} & \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{p}_{2d+1} & \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \mathbf{p}_1 & \mathbf{p}_{2:d} & \mathbf{0} & \mathbf{p}_{d+2:2d} & \mathbf{p}_{2d+1} & \mathbf{p}_{2d+2:3d} \\ 0 & 0_{2:d} & 1 & 0_{d+2:2d} & 0 & 0_{2d+2:3d} \end{bmatrix} \to \begin{bmatrix} * & * & * & * & \sum_{i=1}^m c_{ji}\phi(\mathbf{x}^T\mathbf{a}_{ji}) & * \\ \mathbf{0} & \mathbf{0} & \mathbf{x} & \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{p}_{2d+1} & \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \mathbf{p}_1 & \mathbf{p}_{2:d} & \mathbf{0} & \mathbf{p}_{d+2:2d} & \mathbf{p}_{2d+1} & \mathbf{p}_{2d+2:3d} \\ 0 & 0_{2:d} & 1 & 0_{d+2:2d} & 0 & 0_{2d+2:3d} \end{bmatrix}$$

The proof follows that of Lemma 11. We again ignore the last three rows by setting the corresponding rows in the key, query and values weight matrices to be zero. Let

$$\mathbf{Q}^i = \begin{bmatrix} \mathbf{0} & \mathbf{I}_d \\ \mathbf{0} & \mathbf{0} \end{bmatrix}, \mathbf{K}^i = \begin{bmatrix} [\mathbf{a}_{1i} \ \dots \ \mathbf{a}_{Ni}] & \mathbf{0} \\ \mathbf{0} & \mathbf{0} \end{bmatrix}, \mathbf{V}^i = \begin{bmatrix} [c_{1i} \ \dots \ c_{Ni}] & \mathbf{0} \\ \mathbf{0} & \mathbf{0} \end{bmatrix}$$

We note that for the purpose of this proof, each $\mathbf{a}_i$ has one extra element at the end equal to $-\log(3d-1)$, while the vectors $\mathbf{x}$ will have the last element equal to one. Then we will have

$$\sigma_{\text{S}}((\mathbf{K}^i\mathbf{X})^T(\mathbf{Q}^i\mathbf{X})) = \sigma_{\text{S}}\left(\begin{bmatrix} \mathbf{a}_{ji}^\top & \mathbf{0} \\ \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} \end{bmatrix} \begin{bmatrix} \mathbf{0} & \mathbf{0} & \mathbf{x} & \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} \end{bmatrix} \right)$$

$$= \sigma_{\text{S}}\left(\begin{bmatrix} \mathbf{0} & \mathbf{0} & \mathbf{a}_{ji}^\top\mathbf{x} & \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} \end{bmatrix} \right)$$

$$= \begin{bmatrix} * & * & \phi(\mathbf{x}^T\mathbf{a}_{ji}) & * & * & * \\ * & * & * & * & * & * \\ * & * & * & * & * & * \\ * & * & * & * & * & * \\ * & * & * & * & * & * \\ * & * & * & * & * & * \end{bmatrix}$$

since $\mathbf{a}_{ji}^\top\mathbf{x} = \mathbf{a}_{ji}^\top\mathbf{x} - \log 3d - 1$ and thus $e^{\mathbf{a}_{ji}^\top\mathbf{x}}/(3d - 1 + e^{\mathbf{a}_{ji}^\top\mathbf{x}}) = \phi(\mathbf{a}_{ji}^\top\mathbf{x})$ with a slight abuse of notation over the inner product $\mathbf{a}_{ji}^\top\mathbf{x}$ to account for the extra corrections bias term. Thus,

$$\mathbf{V}\mathbf{X}\sigma_{\text{S}}((\mathbf{K}\mathbf{X})^T(\mathbf{Q}\mathbf{X})) = \begin{bmatrix} * & * & c_{ji}\phi(\mathbf{x}^T\mathbf{a}_{ji}) & * & * & * \\ \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} \end{bmatrix}$$

By summing over all heads and adding the residual we get

$$\begin{bmatrix} * & * & \sum_{i=1}^m c_{ji}\phi(\mathbf{x}^T\mathbf{a}_{ji}) & * & * & * \\ \mathbf{0} & \mathbf{0} & \mathbf{x} & \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{p}_{2d+1} & \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \mathbf{p}_1 & \mathbf{p}_{2:d} & \mathbf{0} & \mathbf{p}_{d+2:2d} & \mathbf{p}_{2d+1} & \mathbf{p}_{2d+2:3d} \\ 0 & 0_{2:d} & 1 & 0_{d+2:2d} & 0 & 0_{2d+2:3d} \end{bmatrix}$$

Finally, we use an extra layer similarly to Lemma 4 to write the result in the desired output. Hence, we get

$$\begin{bmatrix} * & * & * & * & \sum_{i=1}^m c_{ji}\phi(\mathbf{x}^T\mathbf{a}_{ji}) & * \\ \mathbf{0} & \mathbf{0} & \mathbf{x} & \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{p}_{2d+1} & \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \mathbf{p}_1 & \mathbf{p}_{2:d} & \mathbf{0} & \mathbf{p}_{d+2:2d} & \mathbf{p}_{2d+1} & \mathbf{p}_{2d+2:3d} \\ 0 & 0_{2:d} & 1 & 0_{d+2:2d} & 0 & 0_{2d+2:3d} \end{bmatrix}$$

□

Alternative Lemma. We demonstrate an alternative way of encodings functions in the attention mechanism, which has a different complexity tradeoff.

Lemma 12. *Consider an input of the form*

$$\mathbf{X} = \begin{bmatrix} \boldsymbol{x} & \dots & \boldsymbol{x} \\ 0 & \dots & 0 \\ \mathbf{1} - \boldsymbol{e}_1 & \dots & \mathbf{1} - \boldsymbol{e}_d \\ \boldsymbol{e}_1 & \dots & \boldsymbol{e}_d \end{bmatrix}$$

where $\boldsymbol{z} = \mathbf{e}_j \in \mathbb{R}^d$ is an indicator vector and $\mathbf{x} \in \mathbb{R}^d$; then there exists a one layer transformer with 1 head such that

$$\mathrm{Attn}(\mathbf{X}) = \begin{bmatrix} \boldsymbol{x} & \dots & \boldsymbol{x} \\ \sigma(\mathbf{a}_1^\top \boldsymbol{x}) & \dots & \sigma(\mathbf{a}_d^\top \boldsymbol{x}) \\ \mathbf{1} - \boldsymbol{e}_1 & \dots & \mathbf{1} - \boldsymbol{e}_d \\ \boldsymbol{e}_1 & \dots & \boldsymbol{e}_d \end{bmatrix}$$

Proof. Let

$$\mathbf{K} = \begin{bmatrix} \mathbf{a}_1^\top & 0 & -C\boldsymbol{e}_1^\top & \mathbf{0}^\top \\ \vdots & \vdots & \vdots & \vdots \\ \mathbf{a}_d^\top & 0 & -C\boldsymbol{e}_d^\top & \mathbf{0}^\top \\ \mathbf{0}^\top & 0 & \mathbf{0}^\top & \mathbf{0}^\top \end{bmatrix}, \mathbf{Q} = \begin{bmatrix} \mathbf{0}^\top & 0 & \mathbf{0}^\top & \boldsymbol{e}_1^\top \\ \vdots & \vdots & \vdots & \vdots \\ \mathbf{0}^\top & 0 & \mathbf{0}^\top & \boldsymbol{e}_d^\top \end{bmatrix}$$

Hence,

$$\mathbf{KX} = \begin{bmatrix} \mathbf{a}_1^\top \boldsymbol{x} & -C + \mathbf{a}_1^\top \boldsymbol{x} & \dots & -C + \mathbf{a}_1^\top \boldsymbol{x} \\ -C + \mathbf{a}_2^\top \boldsymbol{x} & \mathbf{a}_2^\top \boldsymbol{x} & \dots & -C + \mathbf{a}_2^\top \boldsymbol{x} \\ \vdots & \vdots & \vdots & \vdots \\ -C + \mathbf{a}_d^\top \boldsymbol{x} & -C + \mathbf{a}_d^\top \boldsymbol{x} & \dots & \mathbf{a}_d^\top \boldsymbol{x} \\ 0 & 0 & \dots & 0 \end{bmatrix}, \mathbf{QX} = \mathbf{I}_d$$

After applying softmax we get,

$$\sigma_s((\mathbf{KX})^\top \mathbf{QX}) \approx \begin{bmatrix} \sigma(\mathbf{a}_1^\top \boldsymbol{x}) & 0 & \dots & 0 \\ 0 & \sigma(\mathbf{a}_2^\top \boldsymbol{x}) & \dots & 0 \\ \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & \dots & \sigma(\mathbf{a}_d^\top \boldsymbol{x}) \\ * & 0 & \dots & * \end{bmatrix},$$

for large enough C. Next we set

$$\mathbf{V} = \begin{bmatrix} 0 & 0 & \dots & 0 & 0 \\ 1 & 1 & \dots & 1 & 0 \\ 0 & 0 & \dots & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & \dots & 0 & 0 \end{bmatrix} \mathbf{Q},$$

thus resulting in

$$\mathbf{VX} = \begin{bmatrix} 0 & 0 & \dots & 0 & 0 \\ 1 & 1 & \dots & 1 & 0 \\ 0 & 0 & \dots & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & \dots & 0 & 0 \end{bmatrix} \mathbf{QX} = \begin{bmatrix} 0 & 0 & \dots & 0 & 0 \\ 1 & 1 & \dots & 1 & 0 \\ 0 & 0 & \dots & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & \dots & 0 & 0 \end{bmatrix}.$$

Hence, we get

$$\mathbf{VX}\sigma_s((\mathbf{KX})^\top \mathbf{QX}) = \begin{bmatrix} 0 & \dots & 0 & \\ \sigma(\mathbf{a}_1^\top \boldsymbol{x}) & \dots & \sigma(\mathbf{a}_d^\top \boldsymbol{x}) & \\ 0 & \dots & 0 & \\ \vdots & \vdots & \vdots & \vdots \\ 0 & \dots & 0 & \\ 0 & \dots & 0 & \end{bmatrix},$$

and

$$\mathbf{X} + \mathbf{V}\mathbf{X}\sigma_s((\mathbf{K}\mathbf{X})^\top \mathbf{Q}\mathbf{X}) = \begin{bmatrix} \boldsymbol{x} & \dots & \boldsymbol{x} \\ \sigma(\mathbf{a}_1^\top \boldsymbol{x}) & \dots & \sigma(\mathbf{a}_d^\top \boldsymbol{x}) \\ \mathbf{1} - \boldsymbol{e}_1 & \dots & \mathbf{1} - \boldsymbol{e}_d \\ \boldsymbol{e}_1 & \dots & \boldsymbol{e}_d \end{bmatrix}.$$

□

Corollary 2. *Consider an input of the form*

$$\mathbf{X} = \begin{bmatrix} \boldsymbol{x} & \dots & \mathbf{0} \\ 0 & \dots & 0 \\ \mathbf{1} - \boldsymbol{e}_1 & \dots & \mathbf{1} - \boldsymbol{e}_d \\ \boldsymbol{e}_1 & \dots & \boldsymbol{e}_d \end{bmatrix}$$

where m is the number of sigmoids we use and $\mathbf{e}_i$ is an indicator vector and $\mathbf{x} \in \mathbb{R}^d$; then there exists a 3 layer transformer with 1 head such that

$$\mathrm{Attn}(\mathbf{X}) = \begin{bmatrix} \sum_{i=1}^d \sigma(\mathbf{a}_1^\top \boldsymbol{x}) & \dots & \sum_{i=1}^d \sigma(\mathbf{a}_1^\top \boldsymbol{x}) \\ \mathbf{0} & \dots & \mathbf{0} \end{bmatrix}$$

Proof. Given the input

$$\mathbf{X} = \begin{bmatrix} \boldsymbol{x} & \dots & \mathbf{0} \\ 0 & \dots & 0 \\ \mathbf{1} - \boldsymbol{e}_1 & \dots & \mathbf{1} - \boldsymbol{e}_d \\ \boldsymbol{e}_1 & \dots & \boldsymbol{e}_d \end{bmatrix},$$

we set the query and key matrices as follows:

$$\mathbf{K} = \mathbf{Q} = \begin{bmatrix} \mathbf{0}^\top & 0 & \mathbf{1} & \mathbf{1} \end{bmatrix}.$$

Then, we get

$$(\mathbf{K}\mathbf{X})^\top \mathbf{Q}\mathbf{X} = \begin{bmatrix} d & \dots & d \\ \vdots & \dots & \vdots \\ d & \dots & d \end{bmatrix}.$$

Setting the value matrix to

$$\begin{bmatrix} d\mathbf{I} & \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} \end{bmatrix},$$

we get

$$\mathbf{V}\mathbf{X}\sigma_{\mathrm{S}}((\mathbf{K}\mathbf{X})^\top \mathbf{Q}\mathbf{X}) = \begin{bmatrix} \boldsymbol{x} & \dots & \boldsymbol{x} \\ 0 & \dots & 0 \\ \mathbf{0} & \dots & \mathbf{0} \\ \mathbf{0} & \dots & \mathbf{0} \end{bmatrix}.$$

Hence, the output of the attention layer is:

$$\mathbf{X} + \mathbf{V}\mathbf{X}\sigma_{\mathrm{S}}((\mathbf{K}\mathbf{X})^\top \mathbf{Q}\mathbf{X}) = \begin{bmatrix} 2\boldsymbol{x} & \dots & \boldsymbol{x} \\ 0 & \dots & 0 \\ \mathbf{1} - \boldsymbol{e}_1 & \dots & \mathbf{1} - \boldsymbol{e}_d \\ \boldsymbol{e}_1 & \dots & \boldsymbol{e}_d \end{bmatrix}.$$

Note that using the embeddings in the last rows and a feedforward network can be used to produce the following

$$\begin{bmatrix} \boldsymbol{x} & \dots & \boldsymbol{x} \\ 0 & \dots & 0 \\ \mathbf{1} - \boldsymbol{e}_1 & \dots & \mathbf{1} - \boldsymbol{e}_d \\ \boldsymbol{e}_1 & \dots & \boldsymbol{e}_d \end{bmatrix}.$$

Now, passing this into the transformer of Lemma 12 will result in

$$\mathrm{Attn}(\mathbf{X}) = \begin{bmatrix} \boldsymbol{x} & \dots & \boldsymbol{x} \\ \sigma(\mathbf{a}_1^\top \boldsymbol{x}) & \dots & \sigma(\mathbf{a}_d^\top \boldsymbol{x}) \\ \mathbf{1} - \boldsymbol{e}_1 & \dots & \mathbf{1} - \boldsymbol{e}_d \\ \boldsymbol{e}_1 & \dots & \boldsymbol{e}_d \end{bmatrix}.$$

For the third layer, we set the key and query matrices as follows

$$\mathbf{K} = \mathbf{Q} = \begin{bmatrix} \mathbf{0}^\top & 0 & \mathbf{1} & \mathbf{1} \end{bmatrix}.$$

Then, we get

$$(\mathbf{KX})^\top \mathbf{QX} = \begin{bmatrix} d & \dots & d \\ \vdots & \dots & \vdots \\ d & \dots & d \end{bmatrix}.$$

Setting the value matrix to

$$\begin{bmatrix} \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & d & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} \end{bmatrix},$$

we get

$$\mathbf{VX}\sigma_{\mathrm{S}}((\mathbf{KX})^\top \mathbf{QX}) = \begin{bmatrix} \mathbf{0} & \dots & \mathbf{0} \\ \sum_{i=1}^d \sigma(\mathbf{a}_1^\top \boldsymbol{x}) & \dots & \sum_{i=1}^d \sigma(\mathbf{a}_1^\top \boldsymbol{x}) \\ \mathbf{0} & \dots & \mathbf{0} \\ \mathbf{0} & \dots & \mathbf{0} \end{bmatrix}.$$

Hence, the output of the attention layer is:

$$\mathbf{X} + \mathbf{VX}\sigma_{\mathrm{S}}((\mathbf{KX})^\top \mathbf{QX}) = \begin{bmatrix} \boldsymbol{x} & \dots & \boldsymbol{x} \\ \sum_{i=1}^d \sigma(\mathbf{a}_1^\top \boldsymbol{x}) & \dots & \sum_{i=1}^d \sigma(\mathbf{a}_1^\top \boldsymbol{x}) \\ \mathbf{1} - \boldsymbol{e}_1 & \dots & \mathbf{1} - \boldsymbol{e}_d \\ \boldsymbol{e}_1 & \dots & \boldsymbol{e}_d \end{bmatrix}.$$

Finally, the feedforward layers can be used to move the results to the first row. □

J.2 Matrix Transposition

Lemma 13. *Fix $\epsilon > 0$ and consider an input of the following form*

$$\mathbf{X} = \left[\begin{array}{c|c|c|c} \mathbf{A} & \mathbf{0} & \mathbf{0} & \dots \quad \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{0} & \dots \quad \mathbf{0} \\ \mathbf{p}_{1:d} & \mathbf{p}_{1:d} & \mathbf{p}_{1:d} & \dots \quad \mathbf{p}_{1:d} \\ \mathbf{P}'_1 & \mathbf{P}'_2 & \mathbf{P}'_3 & \dots \quad \mathbf{P}'_d \end{array}\right].$$

where $\mathbf{A} \in \mathbb{R}^{d\times d}$; then there exists transformer-based function block with 4 layers, 1 head and dimensionality $r = 2d + 2\log d = O(d)$ that outputs the following matrix

$$\mathbf{X} = \left[\begin{array}{c|c|c|c} \mathbf{A}' & \mathbf{A}' & \mathbf{A}' & \dots \quad \mathbf{A}' \\ \mathbf{0} & \mathbf{0} & \mathbf{0} & \dots \quad \mathbf{0} \\ \mathbf{p}_{1:d} & \mathbf{p}_{1:d} & \mathbf{p}_{1:d} & \dots \quad \mathbf{p}_{1:d} \\ \mathbf{P}'_1 & \mathbf{P}'_2 & \mathbf{P}'_3 & \dots \quad \mathbf{P}'_d \end{array}\right].$$

where $\mathbf{A}' = \mathbf{A}^\top + \epsilon\mathbf{M}$, for some $\|\mathbf{M}\| \leq 1$.

Proof. We can vectorize the matrix $\mathbf{A}$ into a d^2 dimensional vector using the attention mechanism, as shown in Eq. (10). Notice that once we have the matrix in this form we can implement its transpose with a fixed permutation of the columns of the matrix to get the vectorized form of $\mathbf{A}^\top$. Once we have the transpose in vector form, we matricize it back to get the matrix transform using the attention mechanism. We explain the details of this process below:

Vectorization: We assume that the input is of the following form, where $\mathbf{A}$ is the matrix to be vectorized.

$$\mathbf{X} = \begin{bmatrix} \mathbf{A} & \mathbf{0} & \dots & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \dots & \mathbf{0} \\ \mathbf{p}_{1:d} & \mathbf{p}_{1:d} & \dots & \mathbf{p}_{1:d} \\ \mathbf{P}'_1 & \mathbf{P}'_2 & \dots & \mathbf{P}'_d \end{bmatrix}.$$

Here, $\mathbf{P}'_i$ represents a matrix of d columns, where each column is $\mathbf{p}_i$.

The first layer uses the $\mathbf{p}_{1:d}$ encodings to make d copies of the matrix $\mathbf{A}$, as follows:

$$\mathbf{X} = \begin{bmatrix} \mathbf{A} & \mathbf{0} & \dots & \mathbf{0} \\ \mathbf{A} & \mathbf{A} & \dots & \mathbf{A} \\ \mathbf{p}_{1:d} & \mathbf{p}_{1:d} & \dots & \mathbf{p}_{1:d} \\ \mathbf{P}'_1 & \mathbf{P}'_2 & \dots & \mathbf{P}'_d \end{bmatrix}.$$

The feed forward part of the second layer then uses the encodings $\mathbf{p}'_i$ to vectorize the matrix in the second row block as follows:

$$\mathbf{X} = \begin{bmatrix} \mathbf{A} & \dots & \mathbf{0} \\ \begin{bmatrix} A_{(1,1)} & \dots & A_{(1,d)} \\ \mathbf{0} & \dots & \mathbf{0} \end{bmatrix} & \dots & \begin{bmatrix} A_{(d,1)} & \dots & A_{(d,d)} \\ \mathbf{0} & \dots & \mathbf{0} \end{bmatrix} \\ \mathbf{p}_{1:d} & \dots & \mathbf{p}_{1:d} \\ \mathbf{P}'_1 & \dots & \mathbf{P}'_d \end{bmatrix}. \tag{10}$$

This is achieved, by explicitly defining a neural network that keeps the $i-$th row if the corresponding encoding is $\mathbf{P}'_i$ and place it in the $d+1$ row.

Transposition in the vector form: Once we have the matrix vectorized as the second row block of the scratchpad, the following key and query matrices

$$\mathbf{K} = \begin{bmatrix} \mathbf{0} & \mathbf{0} & \mathbf{I} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{I} \end{bmatrix}, \mathbf{Q} = \begin{bmatrix} \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{I} \\ \mathbf{0} & \mathbf{0} & \mathbf{I} & \mathbf{0} \end{bmatrix},$$

results in the head outputting the following, which is the vectorized form of $\mathbf{A}^\top$ (in the second row block)

$$\mathbf{X}\sigma_{\mathrm{S}}((\mathbf{KX})^\top(\mathbf{QX})) = \begin{bmatrix} * & \dots & * \\ \begin{bmatrix} A_{(1,1)} & \dots & A_{(d,1)} \\ \mathbf{0} & \dots & \mathbf{0} \end{bmatrix} & \dots & \begin{bmatrix} A_{(1,d)} & \dots & A_{(d,d)} \\ \mathbf{0} & \dots & \mathbf{0} \end{bmatrix} \\ \mathbf{P}'_1 & \dots & \mathbf{P}'_d \\ \mathbf{p}_{1:d} & \dots & \mathbf{p}_{1:d} \end{bmatrix}.$$

Then, using the following value matrix gives

$$\mathbf{V} = \begin{bmatrix} \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{I} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} \end{bmatrix},$$

$$\mathbf{VX}\sigma_{\mathrm{S}}((\mathbf{KX})^\top(\mathbf{QX})) = \begin{bmatrix} \mathbf{0} & \dots & \mathbf{0} \\ \begin{bmatrix} A_{(1,1)} & \dots & A_{(d,1)} \\ \mathbf{0} & \dots & \mathbf{0} \end{bmatrix} & \dots & \begin{bmatrix} A_{(1,d)} & \dots & A_{(d,d)} \\ \mathbf{0} & \dots & \mathbf{0} \end{bmatrix} \\ \mathbf{0} & \dots & \mathbf{0} \\ \mathbf{0} & \dots & \mathbf{0} \end{bmatrix},$$

Adding back the $\mathbf{X}$ (see (1)), results in

$$\mathbf{X} + \mathbf{VX}\sigma_{\mathrm{S}}((\mathbf{KX})^\top(\mathbf{QX})) = \begin{bmatrix} \mathbf{A} & \dots & \mathbf{0} \\ \begin{bmatrix} A_{(1,1)} & \dots & A_{(d,1)} \\ \mathbf{0} & \dots & \mathbf{0} \end{bmatrix} & \dots & \begin{bmatrix} A_{(1,d)} & \dots & A_{(d,d)} \\ \mathbf{0} & \dots & \mathbf{0} \end{bmatrix} \\ \mathbf{p}_{1:d} & \dots & \mathbf{p}_{1:d} \\ \mathbf{P}'_1 & \dots & \mathbf{P}'_d \end{bmatrix}.$$

Using the feedforward layers and the encodings $\mathbf{P}'_i$, we get

$$\mathbf{X} = \begin{bmatrix} \mathbf{A} & \dots & \mathbf{0} \\ \begin{bmatrix} A_{(1,1)} & \dots & A_{(d,1)} \\ \mathbf{0} & \dots & \mathbf{0} \end{bmatrix} & \dots & \begin{bmatrix} \mathbf{0} & \dots & \mathbf{0} \\ A_{(1,d)} & \dots & A_{(d,d)} \end{bmatrix} \\ \mathbf{p}_{1:d} & \dots & \mathbf{p}_{1:d} \\ \mathbf{P}'_1 & \dots & \mathbf{P}'_d \end{bmatrix}.$$

Using an attention layer and the first row of encodings, we get

$$\mathbf{X} = \begin{bmatrix} \mathbf{A}^\top & \dots & \mathbf{A}^\top \\ \mathbf{0} & \dots & \mathbf{0} \\ \mathbf{p}_{1:d} & \dots & \mathbf{p}_{1:d} \\ \mathbf{P}'_1 & \dots & \mathbf{P}'_d \end{bmatrix}.$$

J.3 Matrix Multiplication by Linearizing the Softmax

We will show how we can implement matrix multiplication so that it will fit our unified template. To do so, we need to show for example for the result of $\mathbf{A}^\top\mathbf{B}$, where $\mathbf{A} \in \mathbb{R}^{k\times m}$ and $\mathbf{B} \in \mathbb{R}^{k\times n}$ with $k, m, n < d$ we can achieve the following:

$$\left[\begin{array}{cc|cc|c} \mathbf{A} & \mathbf{0} & \mathbf{B} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} \end{array}\right] \rightarrow \left[\begin{array}{cc|cc|cc} * & * & * & * & \mathbf{A}^\top\mathbf{B} & * \\ \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} \end{array}\right]$$

Lemma 14. *Let $\mathbf{A} \in \mathbb{R}^{k\times m}$ and $\mathbf{B} \in \mathbb{R}^{k\times n}$; then for any $\epsilon > 0$ there exists a transformer-based function block with 2 layers, 1 head and dimensionality $r = O(d)$ that outputs the multiplication $\mathbf{A}^T\mathbf{B}^T + \epsilon\mathbf{M}$, for some $\|\mathbf{M}\| \leq 1$.*

Corollary 3. *Let $\mathbf{A} \in \mathbb{R}^{k\times m}$ and $\mathbf{B} \in \mathbb{R}^{k\times n}$; then for any $\epsilon > 0$ there exists a transformer-based function block with 2 layers, 1 head and dimensionality $r = O(d)$ that outputs the multiplication $\mathbf{B}^\top\mathbf{A} + \epsilon\mathbf{M}$, for some $\|\mathbf{M}\| \leq 1$.*

Corollary 4. *Let $\mathbf{A} \in \mathbb{R}^{k\times m}$ and $\mathbf{B} \in \mathbb{R}^{k\times n}$; then for any $\epsilon > 0$ there exists a transformer-based function block with 2 layers, 1 head and dimensionality $r = O(d)$ that outputs the multiplication $\mathbf{B}^\top\mathbf{B} + \epsilon\mathbf{M}$, for some $\|\mathbf{M}\| \leq 1$.*

Corollary 5. *Let $\mathbf{A} \in \mathbb{R}^{k\times m}$ and $\mathbf{B} \in \mathbb{R}^{k\times n}$; then for any $\epsilon > 0$ there exists a transformer-based function block with 2 layers, 1 head and dimensionality $r = O(d)$ that outputs the multiplication $\mathbf{A}^\top\mathbf{A} + \epsilon\mathbf{M}$, for some $\|\mathbf{M}\| \leq 1$.*

We will prove just the first of these results and the rest are a simple corollary of it.

Proof. Let $\mathbf{M} \in \mathbb{R}^{2d\times 2d}$, $\mathbf{A} \in \mathbb{R}^{k\times m}$ and $\mathbf{B} \in \mathbb{R}^{k\times n}$ be the following matrices:

$$\mathbf{M} = \begin{bmatrix} \mathbf{A} & \mathbf{0} & \mathbf{B} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} \end{bmatrix}.$$

The zeros pad the rows and columns to ensure that the matrix M is $2d \times 2d$. Then, consider the input matrix to be of the following form:

$$\mathbf{X} = \begin{bmatrix} \mathbf{M} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{1}\mathbf{1}^\top & \mathbf{0} \\ \mathbf{I} & \mathbf{0} & \mathbf{0} \\ & \mathbf{p}^{(1)} & \\ & \mathbf{p}^{(2)} & \\ \mathbf{0} & \mathbf{1}^T & \mathbf{0} \end{bmatrix}$$

where $\mathbf{1} \in \mathbb{R}^{2d}$ is the all ones vector. The identity matrix $\mathbf{I}$ and the all ones matrix $\mathbf{1}\mathbf{1}^\top$ are part of the design of the input and they are always fixed. For now we ignore the encodings and the last row, by setting the corresponding rows of the key,query and value weight matrices to be zero. These rows will be used to copy the output to the place that we want.

Focusing on the rest of the rows, we set the key and query weight matrices to be

$$\mathbf{K} = \mathbf{I}, \mathbf{Q} = \begin{bmatrix} c\mathbf{I} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & C\mathbf{I} \\ \mathbf{0} & \mathbf{I} & \mathbf{0} \end{bmatrix}, \mathbf{V} = \begin{bmatrix} \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & ne^C \mathcal{D}_d \\ \mathbf{0} & \mathbf{0} & \mathbf{0} \end{bmatrix}$$

where $\mathcal{D}_d \in \mathbb{R}^{2d \times 2d}$ is the diagonal matrix with the first d diagonal elements 1, and the rest 0. Thus we have

$$\begin{aligned} (\mathbf{K}\mathbf{X})^\top \mathbf{Q}\mathbf{X} &= \begin{bmatrix} \mathbf{M} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{1}\mathbf{1}^\top & \mathbf{0} \\ \mathbf{I} & \mathbf{0} & \mathbf{0} \end{bmatrix}^\top \begin{bmatrix} c\mathbf{M} & \mathbf{0} & \mathbf{0} \\ C\mathbf{I} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{1}\mathbf{1}^\top & \mathbf{0} \end{bmatrix} \\ &= \begin{bmatrix} c\mathbf{M}^\top\mathbf{M} & \mathbf{1}\mathbf{1}^\top & \mathbf{0} & \\ C\mathbf{1}\mathbf{1}^\top & \mathbf{0} & \mathbf{0} & \\ & \mathbf{0} & \mathbf{0} & \mathbf{0} \end{bmatrix} \end{aligned}$$

Each of the first $2d$ columns above looks as follows

$$\begin{bmatrix} cz_{1i} & cz_{2i} & \dots & cz_{ni} & C\mathbf{1}^\top & \mathbf{0} \end{bmatrix}$$

After we apply the softmax σ_s per column, we get

$$\sigma_s(cz_{ij}) = \frac{e^{cz_{ij}}}{\sum_{j=1}^n e^{cz_{ij}} + n(e^C + 1)}$$

where $n = 2d$, z_{ij} is the (i, j) element of the matrix $\mathbf{M}^\top\mathbf{M}$. Let $\ell(\cdot)$ be the transformation above then we have

$$\begin{aligned} \mathbf{V}\mathbf{X}\sigma_{\mathrm{S}}((\mathbf{K}\mathbf{X})^\top\mathbf{Q}\mathbf{X}) &= \begin{bmatrix} \mathbf{0} & \mathbf{0} & \mathbf{0} \\ ne^C\mathcal{D}_d & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{0} \end{bmatrix} \begin{bmatrix} \ell(c\mathbf{M}^\top\mathbf{M}) & * & * \\ * & * & * \\ * & * & * \end{bmatrix} \\ &= \begin{bmatrix} \mathbf{0} & \mathbf{0} & \mathbf{0} \\ ne^C\mathcal{D}_d\ell(c\mathbf{M}^\top\mathbf{M}) & * & * \\ \mathbf{0} & \mathbf{0} & \mathbf{0} \end{bmatrix} \\ &\approx \begin{bmatrix} \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \mathbf{1}\mathbf{1}^\top + c\mathbf{M}^\top\mathbf{M} & * & * \\ \mathbf{0} & \mathbf{0} & \mathbf{0} \end{bmatrix} \end{aligned}$$

and by adding back the residual we have

$$\mathbf{X} = \begin{bmatrix} \mathbf{M} & \mathbf{0} & \mathbf{0} \\ \mathbf{1}\mathbf{1}^\top + c\mathbf{M}^\top\mathbf{M} & * & * \\ \mathbf{I} & \mathbf{0} & \mathbf{0} \end{bmatrix}$$

for small enough c and large enough C. This is because

$$\begin{aligned} ne^C \frac{e^{cx_{ij}}}{\sum_{j=1}^n e^{cx_{ij}} + n(e^C + 1)} &= e^{cx_{ij}} \frac{1}{1 + \sum_{j=1}^n e^{cx_{ij} - C - \log n} + n} \\ &= (1 + cx_{ij} + O((cx_{ij})^2))(1 - e^{cx_{ij} - C - \log n} + O(e^{2(cx_{ij} - C - \log n)})) \\ &= (1 + cx_{ij} + O((cx_{ij})^2))(1 - e^{cx_{ij} - C - \log n}) \\ &\approx (1 + cx_{ij}) \end{aligned} \tag{11}$$

Hence by increasing C and decreasing c, the error can be made arbitrarily small. We now use the feedforward layers to perform the following transform

$$\begin{aligned} \mathbf{X} &= \begin{bmatrix} * & * & * \\ \mathbf{M}^\top\mathbf{M} & * & * \\ * & * & * \end{bmatrix} \\ &= \begin{bmatrix} * & * & * & * & * \\ \mathbf{A}^\top\mathbf{A} & \mathbf{0} & \mathbf{A}^\top\mathbf{B} & \mathbf{0} & * \\ \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & * \\ \mathbf{B}^\top\mathbf{A} & \mathbf{0} & \mathbf{B}^\top\mathbf{B} & \mathbf{0} & * \\ \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & * \\ * & * & * & * & * \end{bmatrix} \end{aligned}$$

Now if $\mathbf{p}^{(1)} = [\mathbf{0} \quad \mathbf{0} \quad \mathbf{p}_{2d+1:2d+n} \quad \mathbf{0} \quad \mathbf{0}]$ and $\mathbf{p}^{(2)} = [\mathbf{p}_{1:n} \quad \mathbf{p}_{n+1:d} \quad \mathbf{0} \quad \mathbf{p}_{d+n+1:2d} \quad \mathbf{p}_{2d:3d}]$ we can copy $\mathbf{A}^\top\mathbf{B}$ to the desired place using Lemma 3. □

J.4 Advantage of attention over fully-connected networks

It is possible to implement the functions and overall lexicographic functionality presented in previous sections using fully connected networks, as they are also universal function approximators. However, it is easy to demonstrate a depth separation between attention-based networks and fully connected networks. For example, to compute simple functions like polynomials of x (*e.g.*, x^2), a ReLU network with a depth proportional to $\log(1/\epsilon)$ is required, where ϵ is the quality of approximation, *e.g.*, as showed in (Perekrestenko et al., 2018). In contrast, we have shown how x^2 can be implemented in essentially 2 layers. This simple depth separation argument highlights the constant vs scaling depth required for several functionalities in fully connected networks versus attention-based networks. It is important to note that although these constructions are easy to demonstrate their existence, constructing them is not straightforward. In this work, we provide hardcoded attention layers that precisely do that, making it easier to implement these functionalities in practice.

K FLEQ: Proof of Theorem 2

Each instruction consists of the following tuple: $(\mathbf{p}_a, \mathbf{p}_b, \mathbf{p}_c, \mathbf{p}_{\text{flag}}, \mathbf{p}_m, \mathbf{p}_p)$, and does the following

1. $mem[c] = f_m(mem[a], mem[b])$
2. if $mem[\text{flag}]_{(0,0)} \leq 0$ goto instruction p

Here, locations a, b, and c can contain either scalars, or d-dimensional vectors or $d \times d$ matrices, and $\texttt{mem}[\text{flag}]_{(0,0)}$ is the 1-st entry of $\texttt{mem}[\text{flag}]$ if it is a vector / matrix, else it is $\texttt{mem}[\text{flag}]$ if a scalar.

This can be implemented using the following steps (each may use a separate layer of transformer):

At the beginning of each iteration, the scratchpad starts with storing the pointer to the next instruction $\mathbf{p}_t$.

1. Read the command $(\mathbf{p}_a, \mathbf{p}_b, \mathbf{p}_c, \mathbf{p}_{\text{flag}}, \mathbf{p}_p, \mathbf{p}_m)$ from the location to the scratchpad.
2. Copy the $d \times d$ data at locations a, b to the scratchpad memory $scratchMem$ (assume the data is $d \times d$ even if actually scalar or vector, the f_m implementation will handle that)
3. Copy the data to the i-th function row block using the feed forward layer.
4. Once in the correct row block, $f_m(\texttt{mem}[a], \texttt{mem}[b])$ is computed
5. Feedforward layers copy back the data from i-th row block to the scratchpad memory $scratchMem$.
6. Write result from scratchpad memory to $\mathbf{p}_c$.
7. if $\texttt{mem}[\text{flag}]_{(0,0)} \leq 0$ store $\mathbf{p}_p$ in the scratchpad, else $\mathbf{p}_{t+1}$

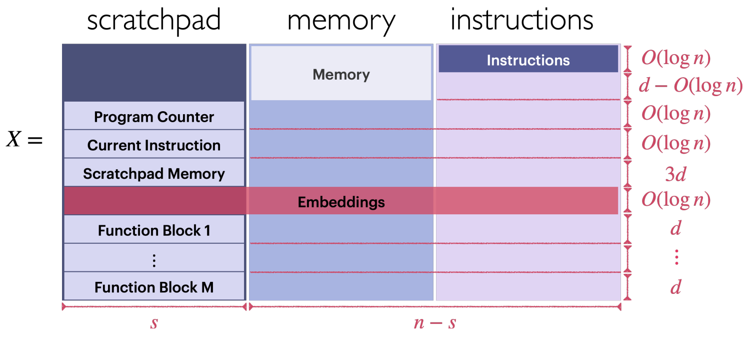

Figure 7: The structure of input $\mathbf{X}$

The structure of the input $\mathbf{X}$ is shown in Figure 7. It has n columns and $O(Md + \log n)$ rows. It is partitioned into 3 column blocks: the Scratchpad block, the Memory block, and the Instructions block. The Memory block is the storage and is the location where all the variables are stored. Each variable can be either a scalar, vector or matrix, as long as the number of rows in it are no larger than d. For example, if a variable is a $d \times d$ matrix, it is stored in d consecutive columns in the block, where each column has length d. The address of this variable is the index of its first column in the input $\mathbf{X}$. The Instructions block contains instructions, where each instruction is a vector of the form

$$\boldsymbol{c} = \begin{bmatrix} \mathbf{p}_a \\ \mathbf{p}_b \\ \mathbf{p}_c \\ \mathbf{p}_m \\ \mathbf{p}_{\text{flag}} \\ \mathbf{p}_p \\ d_h \\ d_w \\ b_{\text{mask}}^{(1)} \\ b_{\text{mask}}^{(2)} \\ b_{\text{mask}}^{(3)} \end{bmatrix},$$

which encodes the following logic:

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

$\mathbf{p}_a, \mathbf{p}_b, \mathbf{p}_c, \mathbf{p}_p$, and $\mathbf{p}_{\text{flag}}$ are all binary ± 1 vectors that point to the locations a, b, c, p, and flag respectively. These are simply the binary representations of the integers a, b, c, p and flag, and hence have length $\log_2 n$ each. Similarly, $\mathbf{p}_m$ is the binary vector representation of the integer m, and hence has length $\log_2 M$, where M is the number of functions we implement. The b_{mask} is mask bit used while writing the output back to memory.

The scratchpad has s columns. The length s depends on the maximum number of columns needed by the function blocks to operate, and can be as low as $O(1)$ for scalar and vector functions, $O(d)$ for matrix functions, and can be as high as $O(d^2)$ if functions like matrix vectorization are one of the M functions. The Scratchpad consists of the following parts:

- The program counter is a row block with $\log_2 n$ rows and s columns and takes the form:

  $$[\mathbf{p}_i \quad \mathbf{p}_i \quad \cdots \quad \mathbf{p}_i.]$$

  This signifies that the current program counter points to the i-th instruction. Using this, the i-th instruction is read into all the s columns of 'Current Instruction' row block.

- The Current Instruction row block has $O(\log n)$ rows and s columns, and each column initially contains the i-th instruction once it is read. Then, the instructions in each column are slightly modified depending on the column index, to read memory blocks pointed to in the instruction. The memory blocks are read into the 'Scratchpad Memory'.

- The Scratchpad Memory is a temporary location where the data is first read into from the Memory column block, before it is moved to the correct function's Function Block, using the function index encoding $\mathbf{p}_m$ in the instruction.

- The encodings row block has $O(\log n)$ rows and n columns, and is used to index every column in the input $\mathbf{X}$. It contains the binary ± 1 vector encodings of the column index for each column. The details of this row block are explained later.

- The Function Blocks are custom transformer blocks that can be added in a plug-n-play manner to the Unified Attention Based Computer depending on what 'elementary' functions the user wants the computer to have access to.

$$\mathbf{X} = \left[\begin{array}{cccc|ccc|ccc}
\mathbf{0} & \mathbf{0} & \ldots & \mathbf{0} & \mathbf{z}_{s+1} & \ldots & \mathbf{z}_{m+s} & \begin{bmatrix}\boldsymbol{c}_{m+s+1}\\ \mathbf{0}\end{bmatrix} & \ldots & \begin{bmatrix}\boldsymbol{c}_{n}\\ \mathbf{0}\end{bmatrix} \\
\hline
\mathbf{p}_t & \mathbf{p}_t & \ldots & \mathbf{p}_t & * & \ldots & * & * & \ldots & * \\
\hline
\boldsymbol{c}_t^1 & \boldsymbol{c}_t^2 & \ldots & \boldsymbol{c}_t^s & * & \ldots & * & * & \ldots & * \\
\hline
\mathbf{z}_{a_t}^1 & \mathbf{z}_{a_t}^2 & \ldots & \mathbf{z}_{a_t}^s & \mathbf{0} & \ldots & \mathbf{0} & \mathbf{0} & \ldots & \mathbf{0} \\
\mathbf{z}_{b_t}^1 & \mathbf{z}_{b_t}^2 & \ldots & \mathbf{z}_{b_t}^s & \mathbf{0} & \ldots & \mathbf{0} & \mathbf{0} & \ldots & \mathbf{0} \\
\mathbf{z}_{c_t}^1 & \mathbf{z}_{c_t}^2 & \ldots & \mathbf{z}_{c_t}^s & \mathbf{0} & \ldots & \mathbf{0} & \mathbf{0} & \ldots & \mathbf{0} \\
\hline
\mathbf{0} & \mathbf{0} & \ldots & \mathbf{0} & \mathbf{p}_{s+1} & \cdots & \mathbf{p}_{m+s} & \mathbf{p}_{m+s+1} & \cdots & \mathbf{p}_n \\
\mathbf{p}_1 & \mathbf{p}_2 & \ldots & \mathbf{p}_s & \mathbf{0} & \ldots & \mathbf{0} & \mathbf{0} & \ldots & \mathbf{0} \\
\hline
\hline
\mathrm{f}_1\mathtt{mem} & \ldots & \ldots & \ldots & * & \ldots & * & \ldots & * & \\
\vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \ldots & * & \\
\mathrm{f}_M\mathtt{mem} & \ldots & \ldots & \ldots & * & \ldots & * & \ldots & * &
\end{array}\right]$$

K.1 Step 1

In this step, we need to copy the t-th instruction, pointed to by the program counter $\mathbf{p}_t$, to the scratchpad's Current Instruction block. We denote the instruction by $\boldsymbol{c}_t$ where

$$\boldsymbol{c}_t = \begin{bmatrix} \mathbf{p}_{a_t} \\ \mathbf{p}_{b_t} \\ \mathbf{p}_{c_t} \\ \mathbf{p}_{\text{flag}_t} \\ \mathbf{p}_{p_t} \\ \mathbf{p}_{m_t} \\ d_h \\ d_w \\ b_{\text{mask}}^{(1)} \\ b_{\text{mask}}^{(2)} \\ b_{\text{mask}}^{(3)} \end{bmatrix}$$

For this step, we only consider the following relevant subset of rows of the matrix $\mathbf{X}$:

$$\mathbf{X} = \left[\begin{array}{cccc|ccc|ccc}
\mathbf{0} & \mathbf{0} & \ldots & \mathbf{0} & * & * & \ldots & \boldsymbol{c}_{m+s+1} & \ldots & \boldsymbol{c}_n \\
\hline
\mathbf{p}_t & \mathbf{p}_t & \ldots & \mathbf{p}_t & * & \ldots & * & * & \ldots & * \\
\hline
\boldsymbol{c}_t^1 & \boldsymbol{c}_t^2 & \ldots & \boldsymbol{c}_t^s & * & \ldots & * & * & \ldots & * \\
\hline
\mathbf{0} & \mathbf{0} & \ldots & \mathbf{0} & \mathbf{p}_{s+1} & \cdots & \mathbf{p}_{m+s} & \mathbf{p}_{m+s+1} & \cdots & \mathbf{p}_n
\end{array}\right]$$

The other rows will not be used or changed during this operation because we can simply set the corresponding rows of the $\mathbf{K}, \mathbf{V}, \mathbf{Q}$ matrices to 0 for all heads and setting the feed forward layers to also pass the corresponding rows unchanged.

At the beginning of execution of each command, the Current Instruction row block would be empty, so the input would look like

$$\mathbf{X} = \left[\begin{array}{cccc|ccc|ccc}
* & * & \ldots & * & * & * & \ldots & \boldsymbol{c}_{m+s+1} & \ldots & \boldsymbol{c}_n \\
\hline
\mathbf{p}_t & \mathbf{p}_t & \ldots & \mathbf{p}_t & * & \ldots & * & * & \ldots & * \\
\hline
\mathbf{0} & \mathbf{0} & \ldots & \mathbf{0} & \mathbf{0} & \ldots & \mathbf{0} & \mathbf{0} & \ldots & * \\
\hline
\mathbf{0} & \mathbf{0} & \ldots & \mathbf{0} & \mathbf{p}_{s+1} & \cdots & \mathbf{p}_{m+s} & \mathbf{p}_{m+s+1} & \cdots & \mathbf{p}_n
\end{array}\right]$$

Then, consider an attention head with the following $\mathbf{K}, \mathbf{Q}, \mathbf{V}$ matrices:

$$\mathbf{K} = \begin{bmatrix}\mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{I}\end{bmatrix}, \mathbf{Q} = \begin{bmatrix}\mathbf{0} & \mathbf{I} & \mathbf{0} & \mathbf{0}\end{bmatrix}, \mathbf{V} = \begin{bmatrix}\mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \mathbf{I} & \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0}\end{bmatrix}$$

This will result in

$$\mathbf{X} = \left[\begin{array}{cccc|ccc|ccc}
* & * & \ldots & * & * & * & \ldots & \boldsymbol{c}_{m+s+1} & \ldots & \boldsymbol{c}_n \\
\hline
\mathbf{p}_t & \mathbf{p}_t & \ldots & \mathbf{p}_t & * & \ldots & * & * & \ldots & * \\
\hline
\boldsymbol{c}_t & \boldsymbol{c}_t & \ldots & \boldsymbol{c}_t & * & \ldots & * & * & \ldots & * \\
\hline
\mathbf{0} & \mathbf{0} & \ldots & \mathbf{0} & \mathbf{p}_{s+1} & \cdots & \mathbf{p}_{m+s} & \mathbf{p}_{m+s+1} & \cdots & \mathbf{p}_n
\end{array}\right].$$

We apply Lemma 10 on the row blocks

$$\left[\begin{array}{cccc|ccc|ccc} \boldsymbol{c}_t & \boldsymbol{c}_t & \dots & \boldsymbol{c}_t & * & \dots & * & * & \dots & * \\ \mathbf{p}_1 & \mathbf{p}_2 & \dots & \mathbf{p}_s & \mathbf{0} & \dots & \mathbf{0} & \mathbf{0} & \dots & \mathbf{0} \end{array}\right]$$

to construct feedforward layers that convert $\boldsymbol{c}_t$ to $\boldsymbol{c}_t^i$, where

$$\boldsymbol{c}_t^i = \begin{bmatrix} \mathbf{p}_{a_t+i} \\ \mathbf{p}_{b_t+i-d} \\ \mathbf{p}_{c_t+i-2d} \\ \mathbf{p}_{\text{flag}_t} \\ \mathbf{p}_{p_t} \\ \mathbf{p}_{m_t} \\ d_\text{h} \\ d_\text{w} \\ b_\text{mask}^{(1)} = 1_{(i \le d_w)} \\ b_\text{mask}^{(2)} = 1_{(i > d)} + 1_{(i \le d + d_w)} - 1 \\ b_\text{mask}^{(3)} = 1_{(i > 2d)} + 1_{(i \le 2d + d_w)} - 1 \end{bmatrix}.$$

Note that the last three elements can be created using the following ReLU:

$$\begin{aligned} b_\text{mask}^{(1)} &= \sigma(2d + d_w - i + 1) - \sigma(2d + d_w - i) \\ b_\text{mask}^{(2)} &= \sigma(i - d) - \sigma(i - d - 1) + \sigma(d + d_\text{w} - i + 1) - \sigma(d + d_w - i) - 1 \\ b_\text{mask}^{(3)} &= \sigma(i - 2d) - \sigma(i - 2d - 1) + \sigma(2d + d_w - i + 1) - \sigma(2d + d_\text{w} - i) - 1. \end{aligned}$$

At the end of this step, we get the following:

$$\mathbf{X} = \left[\begin{array}{cccc|ccc|ccc} \mathbf{0} & \mathbf{0} & \dots & \mathbf{0} & * & * & \dots & \boldsymbol{c}_{m+s+1} & \dots & \boldsymbol{c}_n \\ \hline \mathbf{p}_t & \mathbf{p}_t & \dots & \mathbf{p}_t & * & \dots & * & * & \dots & * \\ \hline \boldsymbol{c}_t^0 & \boldsymbol{c}_t^1 & \dots & \boldsymbol{c}_t^s & * & \dots & * & * & \dots & * \\ \hline \mathbf{0} & \mathbf{0} & \dots & \mathbf{0} & \mathbf{p}_{s+1} & \dots & \mathbf{p}_{m+s} & \mathbf{p}_{m+s+1} & \dots & \mathbf{p}_n \end{array}\right],$$

K.2 Step 2

Use three heads, one each for $\mathbf{p}_a, \mathbf{p}_b$ and $\mathbf{p}_c$.

Using the vectors $\mathbf{p}_{a_t+i}, \mathbf{p}_{b_t+i-d}$, and $\mathbf{p}_{c_t+i-2d}$ we copy the data (using one head each and a similar technique as last step) to get the following in the Scratchpad Memory:

$$\left[\begin{array}{ccccccccc|ccc|ccc} \mathbf{z}_{a_t} & \dots & \mathbf{z}_{a_t+d} & * & \dots & * & * & \dots & * & * & \dots & * & * & \dots & * \\ * & \dots & * & \mathbf{z}_{b_t} & \dots & \mathbf{z}_{b_t+d} & * & \dots & * & * & \dots & * & * & \dots & * \\ * & \dots & * & * & \dots & * & \mathbf{z}_{c_t} & \dots & \mathbf{z}_{c_t+s-2d} & * & \dots & * & * & \dots & * \end{array}\right]$$

Using the mask bits at the end of $\boldsymbol{c}_t^i$, we get

$$\left[\begin{array}{cccccccccccc|cc|cc} \mathbf{z}_{a_t} & \dots & \mathbf{z}_{a_t+d_\text{w}-1} & \mathbf{0} & \mathbf{z}_{b_t} & \dots & \mathbf{z}_{b_t+d_\text{w}-1} & \mathbf{0} & \mathbf{z}_{c_t} & \dots & \mathbf{z}_{c_t+d_\text{w}-1} & \mathbf{0} & \mathbf{0} & \dots & \mathbf{0} & \dots \\ \mathbf{0} & \dots & \mathbf{0} & \mathbf{0} & \mathbf{0} & \dots & \mathbf{0} & \mathbf{0} & \mathbf{0} & \dots & \mathbf{0} & \mathbf{0} & \mathbf{0} & \dots & \mathbf{0} & \dots \\ \mathbf{0} & \dots & \mathbf{0} & \mathbf{0} & \mathbf{0} & \dots & \mathbf{0} & \mathbf{0} & \mathbf{0} & \dots & \mathbf{0} & \mathbf{0} & \mathbf{0} & \dots & \mathbf{0} & \dots \end{array}\right] \tag{12}$$

$$\begin{aligned} \mathbf{z}_i[1:d] &= \sigma(\mathbf{z}_i[1:d] - C(1 - b_\text{mask}^{(1)})\mathbf{1}) - \sigma(-\mathbf{z}_i[1:d] - C(1 - b_\text{mask}^{(1)})\mathbf{1}) \\ &\quad + \sigma(\mathbf{z}_i[d+1:2d] - C(1 - b_\text{mask}^{(2)})\mathbf{1}) - \sigma(-\mathbf{z}_i[d+1:2d] - C(1 - b_\text{mask}^{(1)})\mathbf{1}) \\ &\quad + \sigma(\mathbf{z}_i[2d+1:3d] - C(1 - b_\text{mask}^{(1)})\mathbf{1}) - \sigma(-\mathbf{z}_i[2d+1:3d] - C(1 - b_\text{mask}^{(1)})\mathbf{1}), \\ \mathbf{z}_i[d+1:3d] &= \mathbf{0}, \end{aligned}$$

where C is a large positive constant.

Using the same mask bits, we also mask the row containing the output data pointers for c:

$$\left[\begin{array}{ccccccccccc|ccc|ccc} \mathbf{0} & \dots & \mathbf{0} & \mathbf{0} & \dots & \mathbf{0} & \mathbf{p}_{c_t} & \dots & \mathbf{p}_{c_t+d_w-1} & \mathbf{0} & \dots & \mathbf{0} & \mathbf{0} & \dots & \mathbf{0} & \mathbf{0} & \dots & \mathbf{0} \end{array}\right] \tag{13}$$

K.3 STEP 3

The following feedforward ReLU layer can move the data to the correct function blocks:

$$\begin{aligned}\mathrm{f}_k\texttt{mem}[1:d_h] = (&\sigma(\mathbf{z}[1:d_h] - C((1 - b^{(1)}_{\text{mask}} - b^{(2)}_{\text{mask}})\mathbf{1} + \log M - \mathbf{p}_k^\top \mathbf{p}_m)) \\ &- \sigma(-\mathbf{z}[1:d_h] - C((1 - b^{(1)}_{\text{mask}} - b^{(2)}_{\text{mask}})\mathbf{1} + \log M - \mathbf{p}_k^\top \mathbf{p}_m))),\end{aligned}$$

where C is a large positive constant.

K.4 STEP 4

Each of the M functions have their own attention heads, which are constructed to be copies of their transformer based function blocks. The results after the attention are written back into their respective row blocks. Since the row blocks are separate, the feedforward layers of each of the transformer based function blocks also work in parallel to store the final results in the respective row blocks.

K.5 STEP 5

Similar to Step 3 we use the following feedforward ReLU layer to move the data from the function block back into the scratchpad memory

$$\begin{aligned}\mathbf{z}[1:d_h] = \mathbf{z}[1:d_h] + \sum_{k=1}^{M} \Big(&\sigma((\mathrm{f}_k\texttt{mem}[1:d_h] - \mathbf{z}[1:d_h]) - C((1 - b^{(3)}_{\text{mask}})\mathbf{1} + \log M - \mathbf{p}_k^\top \mathbf{p}_m)) \\ &-\sigma(-(\mathrm{f}_k\texttt{mem}[1:d_h] - \mathbf{z}[1:d_h]) - C((1 - b^{(3)}_{\text{mask}})\mathbf{1} + \log M - \mathbf{p}_k^\top \mathbf{p}_m))\Big),\end{aligned}$$

where C is a large positive constant.

K.6 STEP 6

For this step we focus on the encoding row block, memory storage row block and the following rows in the input (see (13), (12)):

$$\left[\begin{array}{cccccccccccc|ccc|ccc}
\mathbf{0} & \dots & \mathbf{0} & \mathbf{0} & \dots & \mathbf{0} & \mathbf{0} & \dots & \mathbf{0} & \mathbf{0} & \dots & \mathbf{0} & \mathbf{z}_{s+1} & \dots & \mathbf{z}_{m+s} & \begin{bmatrix}\boldsymbol{c}_{m+s+1}\\ \mathbf{0}\end{bmatrix} & \dots & \begin{bmatrix}\boldsymbol{c}_{n}\\ \mathbf{0}\end{bmatrix} \\ \hline
\mathbf{0} & \dots & \mathbf{0} & \mathbf{0} & \dots & \mathbf{0} & \mathbf{z}^{\text{new}}_{c_t} & \dots & \mathbf{z}^{\text{new}}_{c_t+d_w} & \mathbf{0} & \dots & \mathbf{0} & \mathbf{0} & \dots & \mathbf{0} & \mathbf{0} & \dots & \mathbf{0} \\ \hline
\mathbf{0} & \dots & \mathbf{0} & \mathbf{0} & \dots & \mathbf{0} & \mathbf{p}_{c_t} & \dots & \mathbf{p}_{c_t+d_w} & \mathbf{0} & \dots & \mathbf{0} & \mathbf{0} & \dots & \mathbf{0} & \mathbf{0} & \dots & \mathbf{0} \\ \hline
\mathbf{0} & \dots & \mathbf{0} & \mathbf{0} & \dots & \mathbf{0} & \mathbf{0} & \dots & \mathbf{0} & \mathbf{0} & \dots & \mathbf{0} & \mathbf{p}_{s} & \dots & \mathbf{p}_{m-1} & \mathbf{p}_{m} & \dots & \mathbf{p}_{n-1}
\end{array}\right]$$

We set the Key and Query matrices as follows:

$$\mathbf{K} = \mathbf{Q} = \begin{bmatrix}\mathbf{0}\\ \mathbf{0}\\ \mathbf{I}\\ \mathbf{I}\end{bmatrix}.$$

$$\mathbf{V} = \begin{bmatrix}\mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0}\\ \mathbf{I} & \mathbf{I} & \mathbf{0} & \mathbf{0}\\ \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0}\\ \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0}\end{bmatrix}$$

$$\begin{aligned}&\mathbf{V}\mathbf{X}\sigma_{\text{S}}((\mathbf{K}\mathbf{X})^\top \mathbf{Q}\mathbf{X}) \\ &= \left[\begin{array}{ccccc|cccccccc|c}
\dots & \mathbf{0} & \dots & \mathbf{0} & \dots & \mathbf{0} & \dots & \mathbf{0} & \mathbf{0} & \dots & \mathbf{0} & \mathbf{0} & \dots & \dots \\ \hline
\dots & \frac{\boldsymbol{d}^{\text{new}}_{c_t}+\boldsymbol{d}_{c_t}}{2} & \dots & \frac{\boldsymbol{d}^{\text{new}}_{c_t+d_w}+\boldsymbol{d}_{c_t+d_w}}{2} & \dots & \boldsymbol{d}_0 & \dots & \boldsymbol{d}_{c_t-1} & \frac{\boldsymbol{d}^{\text{new}}_{c_t}+\boldsymbol{d}_{c_t}}{2} & \dots & \frac{\boldsymbol{d}^{\text{new}}_{c_t+d_w}+\boldsymbol{d}_{c_t+d_w}}{2} & \boldsymbol{d}_{c_t+d_w+1} & \dots & \dots \\ \hline
\dots & \mathbf{0} & \dots & \mathbf{0} & \dots & \mathbf{0} & \dots & \mathbf{0} & \mathbf{0} & \dots & \mathbf{0} & \mathbf{0} & \dots & \dots \\ \hline
\dots & \mathbf{0} & \dots & \mathbf{0} & \dots & \mathbf{0} & \dots & \mathbf{0} & \mathbf{0} & \dots & \mathbf{0} & \mathbf{0} & \dots & \dots
\end{array}\right.\end{aligned}$$

Finally, we use the feedforward layers similar to the proof of Lemma 4 to write back $[\boldsymbol{d}^{\text{new}}_{c_t} \ \dots \ \boldsymbol{d}^{\text{new}}_{c_t+d_w}]$ to the correct rows.

K.7 STEP 7

This step is identical to Appendix B.3.

L ERROR ANALYSIS

In all of this section we assume that each element of the input matrix $\mathbf{X}$ has values v_i bounded by some constant G, *i.e.*, $|v_i| \leq G$.

The error in the `read/write` operation. The positional encodings as we have already mentioned have the following properties: $\mathbf{p}_i$ is an $\log(n)$ dimensional ± 1 vector which is the binary representation of i with -1 in the place of 0. Hence, we have $\mathbf{p}_i^\top \mathbf{p}_i = \log(n)$ and each $\mathbf{p}_i^\top \mathbf{p}_j < \log(n)$ for $i \neq j$.

Each time a copy is implemented from one column to another, we create a permutation matrix (a matrix of zeros and ones) which then multiplies the input matrix $\mathbf{X} \in \mathbb{R}^{d \times n}$ from the right and results in permutations of the column space. We thus focus on just one column of the $n \times n$ matrix that is created after we apply the softmax. Let $\mathbf{z}$ be this column of the matrix, ideally we want to output in one position 1 and in the rest 0. In the place that we want to output 1, say the $a-$th position, we have the inner product $\mathbf{z}_a = \mathbf{p}_i^\top \mathbf{p}_i$ for some $i \in [n]$. The rest of the elements in the same column would be $\mathbf{z}_b \leq \mathbf{p}_i^\top \mathbf{p}_j$ for $i \neq j$ and $a \neq b$. Then,

$$\begin{aligned}[\sigma_{\mathrm{S}}((\mathbf{K}\mathbf{X})^\top \mathbf{Q}\mathbf{X})]_{i,i} &= \frac{e^{\lambda \mathbf{p}_i^\top \mathbf{p}_i}}{e^{\lambda \mathbf{p}_i^\top \mathbf{p}_i} + \sum_{j \neq i} e^{\lambda \mathbf{p}_i^\top \mathbf{p}_j}} \\ &= \frac{1}{1 + \sum_{j \neq i} e^{\lambda \mathbf{p}_i^\top \mathbf{p}_j} / e^{\lambda \mathbf{p}_i^\top \mathbf{p}_i}}\end{aligned}$$

Since $\lambda \mathbf{p}_i^\top \mathbf{p}_j < \lambda \mathbf{p}_i^\top \mathbf{p}_i - \lambda$ for $i \neq j$, we have that

$$\begin{aligned}[\sigma_{\mathrm{S}}((\mathbf{K}\mathbf{X})^\top \mathbf{Q}\mathbf{X})]_{i,i} &\geq \frac{1}{1 + ne^{-\lambda}} \\ &\geq \frac{1}{1 + e^{\log n - \lambda}} \\ &\geq 1 - \frac{e^{\log n - \lambda}}{1 + e^{\log n - \lambda}} \\ &\geq 1 - e^{\log n - \lambda}\end{aligned}$$

Thus, for $i \neq j$, $[\sigma_{\mathrm{S}}((\mathbf{K}\mathbf{X})^\top \mathbf{Q}\mathbf{X})]_{i,j} \leq e^{\log n - \lambda}$. This implies that there exist ϵ_i, $i = 1, \ldots, n$ such that

$$\begin{aligned}\mathbf{z}_a &= 1 - \varepsilon_a, \text{ for some } \varepsilon_a \leq e^{\log n - \lambda} \\ \mathbf{z}_b &= \varepsilon_b \text{ for } b \neq a \text{ and for some } \varepsilon_b \leq e^{\log n - \lambda}\end{aligned}$$

Hence, we have that

$$\mathbf{z} = \mathbf{z}^* + \varepsilon$$

where $\mathbf{z}^*$ is the targeted vector and ε is the vector containing the errors $\varepsilon_a, \varepsilon_b$.

Now let $\mathbf{x}_i$ be the $i-$th row of the input matrix $\mathbf{X}$, then we have

$$\begin{aligned}\mathbf{X}\mathbf{z} &= \mathbf{X}\mathbf{z}^* + \mathbf{X}\varepsilon \\ &= \mathbf{X}\mathbf{z}^* + \begin{bmatrix} \langle \mathbf{x}_1, \varepsilon \rangle \\ \vdots \\ \langle \mathbf{x}_d, \varepsilon \rangle \end{bmatrix}\end{aligned}$$

In the general case that all the columns will change, let $\mathbf{P} = \sigma_{\mathrm{S}}((\mathbf{K}\mathbf{X})^\top \mathbf{Q}\mathbf{X})$ and $\mathbf{P}^*$ be the targeted matrix then we have that

$$\mathbf{X}\mathbf{P} = \mathbf{X}\mathbf{P}^* + \mathbf{X}\mathbf{E}$$

where $\mathbf{E} = [\varepsilon_1 \quad \dots \quad \varepsilon_n]$ is the matrix containing all the errors and so

$$\begin{aligned} \|\mathbf{XP} - \mathbf{XP}^*\| &= \max_{1 \le j \le n} \sum_{i=1}^{d} |\langle \mathbf{x}_i, \varepsilon_j \rangle| \\ &\le Gn^2 d e^{\log n - \lambda} \\ &\le e^{\log Gdn^3 - \lambda} \end{aligned}$$

Thus, if $\lambda > \log \dfrac{Gdn^3}{\epsilon}$ we have that

$$\|\mathbf{XP} - \mathbf{XP}^*\| \le \epsilon$$

The error in Matrix Multiplication . This error has already been calculated in Appendix J.3, however we explicitly define it here as follows:

$$\begin{aligned} ne^C \frac{e^{cx_{ij}}}{\sum_{j=1}^n e^{cx_{ij}} + n(e^C + 1)} &= e^{cx_{ij}} \frac{1}{1 + \sum_{j=1}^n e^{cx_{ij} - C - \log n} + n} \\ &= (1 + cx_{ij} + O((cx_{ij})^2))(1 - e^{cx_{ij} - C - \log n} + O(e^{2(cx_{ij} - C - \log n)})) \end{aligned}$$

Let $c = \frac{\epsilon_1}{C_1 G}$ for some constant C_1 and $C = \log \dfrac{C_2}{\epsilon_2}$ for some C_2 then we have

$$\begin{aligned} A &= ne^C \frac{e^{cx_{ij}}}{\sum_{j=1}^n e^{cx_{ij}} + n(e^C + 1)} \\ &= e^{cx_{ij}} \frac{1}{1 + \sum_{j=1}^n e^{cx_{ij} - C - \log n} + n} \\ &= (1 + cx_{ij} + \frac{\epsilon_1^2 x_{ij}^2}{G^2})(1 - \frac{e^{cx_{ij}\epsilon_2}}{n} + \frac{e^{2cx_{ij}}\epsilon_2^2}{n^2}) \\ &= (1 + cx_{ij})(1 - \frac{e^{cx_{ij}\epsilon_2}}{n} + \frac{e^{2cx_{ij}}\epsilon_2^2}{n^2}) + \frac{\epsilon_1^2 x_{ij}^2}{G^2}(1 - \frac{e^{cx_{ij}}\epsilon_2}{n} + \frac{e^{2cx_{ij}}\epsilon_2^2}{n^2}) \end{aligned}$$

Thus,

$$\begin{aligned} |A - (1 + cx_{ij})| &= |-(1 + cx_{ij})\frac{e^{cx_{ij}\epsilon_2}}{n} + \frac{e^{2cx_{ij}}\epsilon_2^2}{n^2} + \frac{\epsilon_1^2 x_{ij}^2}{G^2}(1 - \frac{e^{cx_{ij}}\epsilon_2}{n} + \frac{e^{2cx_{ij}}\epsilon_2^2}{n^2})| \\ &\le \epsilon_1^2(\frac{e^{\epsilon_1/C_1}\epsilon_2}{n} + 2\frac{e^{2\epsilon_1/C_1}\epsilon_2^2}{n^2}) + \frac{e^{\epsilon_1/C_1}\epsilon_2}{n}(1 + \frac{\epsilon_1}{C_1}) \\ &\le 4\frac{e^{\epsilon_1/C_1}\epsilon_2}{n} \end{aligned}$$

Hence if $\epsilon_2 = \epsilon/4$ and $\epsilon_1 = C_1 \log(n\epsilon)$ we have that the total error is less than ϵ.

Function approximation. The error in Lemma 11 is an immediate consequence of Theorem 3 and it is proportional to $1/\sqrt{m}$, where m is the number of heads we are using.

Accumulation of error after T operations. Fix an $\epsilon > 0$ and assume that in the $t-$th iteration the input is $\mathbf{X}_t = \mathbf{X}_t^* + \epsilon_t \mathbf{M}_t$, where $\mathbf{X}_t^*$ is the ideal input $0 < \epsilon_t < \dfrac{t\epsilon}{T}$ and $\mathbf{M}_t$ is a matrix such that $\|\mathbf{M}_t\| \le 1$, we will show that $\mathbf{X}_{t+1} = \mathbf{X}_{t+1}^* + \epsilon_{t+1}\mathbf{M}_{t+1}$, where $\mathbf{X}_{t+1}^*$ is the ideal input, $0 < \epsilon_{t+1} < \dfrac{(t+1)\epsilon}{T}$ and $\mathbf{M}_{t+1}$ is a matrix such that $\|\mathbf{M}_{t+1}\| \le 1$.

- Matrix Multiplication with a matrix $\mathbf{A}$, $\|\mathbf{A}\| \le 1$[2] will have the following result:

$$\mathbf{AX}_t + \epsilon' = \mathbf{AX}_t^* + \epsilon_t \mathbf{AM}_t + \epsilon' \mathbf{M}' = \mathbf{X}_{t+1}^* + (\epsilon_t + \epsilon')\mathbf{M}_{t+1}$$

[2] Notice that this can be assumed without loss of generality, since we can normalize all the errors with the maximum norm of a matrix to the power of T.

where ϵ' is controlled by the constants we use in the design of the function block and $\mathbf{M}_{t+1}$ is some matrix with $\|\mathbf{M}_{t+1}\| \leq 1$. If now $\epsilon' < \frac{\epsilon}{T}$, our claim follows.

- `Read/Write` operations will result to an error of

  $$\mathbf{X}_t\mathbf{P} = \mathbf{X}_t\mathbf{P}^* + \epsilon'\mathbf{M}' = \mathbf{X}_t^*\mathbf{P}^* + \epsilon_t\mathbf{M}_t\mathbf{P}^* + \epsilon'\mathbf{M}'$$

  Notice that as before, since $\|\mathbf{M}'\| \leq 1$ and $\|\mathbf{M}_t\mathbf{P}^*\| \leq 1$ and thus we have $\mathbf{X}_{t+1} = \mathbf{X}_t\mathbf{P} = \mathbf{X}_{t+1}^* + \epsilon_{t+1}\mathbf{M}_{t+1}$, where $\epsilon_{t+1} = \epsilon_t + \epsilon'$. Again if $\epsilon' \leq \frac{\epsilon}{T}$ the result follows.

- The result for function approximation follows in a similar way.

M A Basic Calculator

We show that the FLEQ transformer introduced in the previous section, can be used to build a simple calculator. This transformer consists of six transformer-based function blocks that implement addition, substraction, multiplication, percentage, division and square root. The formal statement is written as below.

Theorem 4. *There exists a transformer with* 12 *layers,* m *heads and dimensionality* $O(\log n)$ *that uses the Unified Attention Based Computer framework in Section C.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.*

Remark 2. *In the proof of this theorem, we use Lemma 11 to approximate the square root and the inversion function. That lemma provides error guarantees in terms of the number of heads* m*. We prove Corollary 2 in the appendix which provides equivalent error guarantees, but where the error decreases with the dimension* d *of the transformer. Depending on the design choices of the transformer, either of the results can be used, and the calculator's error guarantee will also change accordingly.*

We show how one can implement a calculator in our FLEQ framework in Alg. 7.

Algorithm 7 A sample program for executing a basic calculator functionality. The following algorithm performs $\frac{\sqrt{1/(((a+b)-c)\cdot d)}}{100}$

```
Require: $\texttt{mem}[p] = a, \texttt{mem}[q] = b, \texttt{mem}[r] = c, \texttt{mem}[s] = d.$   { Location of the inputs.}
$\texttt{mem}[t] = f_{\text{add}}(\texttt{mem}[p], \texttt{mem}[q])$   $\{\texttt{mem}[t] = a + b.\}$
$\texttt{mem}[t] = f_{\text{sub}}(\texttt{mem}[t], \texttt{mem}[r])$   $\{\texttt{mem}[t] = (a + b) - c.\}$
$\texttt{mem}[t] = f_{\text{mul}}(\texttt{mem}[t], \texttt{mem}[s])$   $\{\texttt{mem}[t] = ((a + b) - c) * d.\}$
$\texttt{mem}[t] = f_{\text{inv}}(\texttt{mem}[t])$   $\{\texttt{mem}[t] = 1/((a + b) - c) * d.\}$
$\texttt{mem}[t] = f_{\text{sqrt}}(\texttt{mem}[t])$   $\{\texttt{mem}[t] = \sqrt{1/((a + b) - c) * d}.\}$
$\texttt{mem}[t] = f_{\text{perc}}(\texttt{mem}[t])$   $\{\texttt{mem}[t] = \frac{\sqrt{1/((a+b)-c)*d}}{100}.\}$
```

Looking at the algorithm, it is clear that for proving the theorem above, it is sufficient to implement the 6 functions (addition, subtraction, multiplication, inversion, square root and percentage) using the transformer-based function blocks defined in Definition 1. We start with two lemmas, which can be proved by constructing transformers that add and subtract in a similar way to the OISC transformer constructed in Appendix C.1.

Lemma 15 (`addition`)**.** *There exists a transformer-based function block with 3 layers, 1 head and dimensionality* $O(1)$ *which can implement* $f(a, b) = a + b$*.*

Proof. Consider the input in the form of Equation (9)

$$\mathbf{X} = \left[\begin{array}{cc|cc|cc} a & \mathbf{0} & b & \mathbf{0} & 0 & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \mathbf{p}_{2d+1} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{p}_{2:d} & \mathbf{p}_{d+1} & \mathbf{p}_{d+2:2d} & \mathbf{p}_{2d+1} & \mathbf{p}_{2d+2:3d} \\ 1 & \mathbf{0} & 0 & \mathbf{0} & 0 & \mathbf{0} \end{array}\right]$$

We can perform the following transformation

$$\begin{aligned} \left[\begin{array}{cc|cc|cc} a & \mathbf{0} & b & \mathbf{0} & 0 & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} \end{array}\right] &\rightarrow \left[\begin{array}{cc|cc|cc} a & \mathbf{0} & b & \mathbf{0} & 0 & \mathbf{0} \\ a & \mathbf{0} & b & \mathbf{0} & 0 & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} \end{array}\right] \\ &\rightarrow \left[\begin{array}{cc|cc|cc} a & \mathbf{0} & 0 & \mathbf{0} & 0 & \mathbf{0} \\ 0 & \mathbf{0} & b & \mathbf{0} & 0 & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} \end{array}\right] \\ &\rightarrow \left[\begin{array}{cc|cc|cc} a+b & \mathbf{0} & 0 & \mathbf{0} & 0 & \mathbf{0} \\ 0 & \mathbf{0} & 0 & \mathbf{0} & 0 & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} \end{array}\right] \\ &\rightarrow \left[\begin{array}{cc|cc|cc} a+b & \mathbf{0} & 0 & \mathbf{0} & a+b & \mathbf{0} \\ 0 & \mathbf{0} & b & \mathbf{0} & 0 & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} \end{array}\right] \end{aligned}$$

The first and second step are implemented with one feed-forward layer each. The third step with the Appendix B.2. We have ignored the last three rows since we don't change them and we only use them for the last step. □

Note that in `addition` as well as the rest of the operations in this proof, softmax leads to an extra error, which can be driven arbitrarily close to 0 by increasing its temperature.

Lemma 16 (`subtraction`)**.** *There exists a transformer-based function block with 3 layers, 1 head and dimensionality $O(1)$ which can implement $f(a,b) = a - b$.*

This lemma can be proved in the exact same way as the previous one. In addition, we can use the theory presented in Lemma 14 to get the following corollaries:

Corollary 6 (`multiplication`)**.** *There exists a transformer-based function block with 2 layers, 1 head and dimensionality $O(d)$ which can implement $f(a,b) = ab$.*

Corollary 7 (`percentage`)**.** *There exists a transformer-based function block with 2 layers, 1 head and dimensionality $O(1)$ which can implement $f(a) = a/100 = a * 0.01$.*

To implement inversion function, we first show that we can approximate inversion with threshold activations, then we can easily conclude that we can also approximate it with sigmoids.

Lemma 17. *Given two constants $\epsilon, \delta \in [0,1]$, there exists a 1 hidden layer neural network f with threshold activation and d activations in the hidden layer, such that*

$$\forall x \in [-C, -\delta] \cup [\delta, C], \left| f(x) - \frac{1}{x} \right| \leq \epsilon,$$

as long as $d = \Omega(\frac{\log(1/(\epsilon\delta))}{\epsilon\delta} + \log C)$.

Proof. We partition $[\delta, C]$ into the following intervals

$$[\delta, \delta(1+\epsilon\delta)), [\delta(1+\epsilon\delta), \delta(1+\epsilon\delta)(1+\epsilon\delta(1+\epsilon\delta))) \ldots, [a_i, a_i(1+\epsilon a_i)), \ldots,$$

that is, if an interval begins at a, then it ends at $a(1+\epsilon a)$. Note that for any point $x \in [a_i, a_i(1+\epsilon a_i))$

$$\begin{aligned} \left| \frac{1}{x} - \frac{1}{a_i} \right| &= \frac{1}{a_i} - \frac{1}{x} \\ &< \frac{1}{a_i} - \frac{1}{a_i(1+\epsilon a_i)} \\ &= \frac{\epsilon}{1+\epsilon a_i} < \epsilon. \end{aligned}$$

Hence two output activations of the form $\frac{1}{a_i}1_{x\geq a_i} - \frac{1}{a_i}1_{x<a_i(1+\epsilon a_i)}$ can be used to approximate $\frac{1}{x}$ in $[a_i, a_i(1+\epsilon a_i))$.

Thus, all that remains is to compute the number of such intervals, and using that we get the number of output activations in the hidden layer. Towards that end, if the i-th interval begins at a_i,

$$a_i = a_{i-1}(1+\epsilon a_{i-1}) \geq a_{i-1}(1+\epsilon\delta) = \delta(1+\epsilon\delta)^{i-2}.$$

Hence,

$$\forall i \geq 2 + \frac{\log 1/(\epsilon\delta)}{\log(1+\epsilon\delta)}, a_i \geq \frac{1}{\epsilon}.$$

Noting that $\log(1+\epsilon\delta) > \frac{\epsilon\delta}{2}$ for $\epsilon, \delta \in [0,1]$, we get that

$$\forall i \geq 2 + \frac{2\log 1/(\epsilon\delta)}{\epsilon\delta}, a_i \geq 1.$$

Once we have that $a_i \geq \frac{1}{\epsilon}$, the number of further partitions needed to reach C would be $O(\log C)$ as shown below:

$$a_j = a_{j-1}(1+\epsilon a_{j-1}) \geq a_{j-1}\left(1+\epsilon\frac{1}{\epsilon}\right) = 2a_{j-1}.$$

Hence, the total number of partitions needed is $O(\frac{\log(1/(\epsilon\delta))}{\epsilon\delta} + \log C)$.

We can similarly approximate $1/x$ on $[-C, -\delta]$ with the same number of output activations. □

Lemma 18. *Given $\epsilon, \delta \in [0,1]$, and $C \geq 1$ there exists a function f of the form $f(x) = \sum_{i=1}^{m} c_i\phi(w_i x + b_i)$, where ϕ is the sigmoid function, such that*

$$\forall x \in [\delta, C], \left|f(x) - \frac{1}{x}\right| \leq \epsilon,$$

as long as $d = \Omega\left(\frac{\log(1/(\epsilon\delta))}{\epsilon\delta} + \log C\right)$.

We can use this lemma along with the result presented in Lemma 11 to get the following corollary:

Corollary 8 (`inversion`)**.** *There exists a transformer-based function block with 3 layers and m heads which can implement $f(a) = \frac{1}{a}$ up to error $\tilde{O}(\frac{1}{\sqrt{m}})$ for all $a \in [\tilde{\Omega}(\frac{1}{\sqrt{m}}), \tilde{O}(e^m)]$.*

Note that using Corollary 6 (multiplication) and Corollary 8 (inversion), the operation of division can be implemented as well. Next, we move on to showing the way of implementing square root. Similarly with division we get the following lemmas for square root.

Lemma 19. *Given $\epsilon \in [0,1]$, there exists a 1 hidden layer neural network f with threshold activation and d activations in the hidden layer, such that*

$$\forall x \in [0, C], \left|f(x) - \sqrt{x}\right| \leq \epsilon,$$

as long as $d = \Omega(\frac{\sqrt{C}}{\epsilon})$.

Proof. We partition $[0, C]$ into the following intervals

$$[0, \epsilon^2)), [\epsilon^2, 4\epsilon^2)\ldots, [i^2\epsilon^2, (i+1)^2\epsilon^2), \ldots.$$

Note that for any point $x \in [i^2\epsilon^2, (i+1)^2\epsilon^2)$

$$|\sqrt{x} - \sqrt{i^2\epsilon^2}| < \sqrt{(i+1)^2\epsilon^2} - \sqrt{i^2\epsilon^2} = \epsilon.$$

Hence two output activations of the form $i\epsilon 1_{x\geq i^2\epsilon^2} - i\epsilon 1_{x<(i+1)^2\epsilon^2}$ can be used to approximate $\sqrt{x}$ in $[i^2\epsilon^2, (i+1)^2\epsilon^2)$.

Thus, all that remains is to compute the number of such intervals, and using that we get the number of output activations in the hidden layer. It is easy to see that the total number of intervals needed would be $\frac{\sqrt{C}}{\epsilon}$. □

Lemma 20. *Given $\epsilon \in [0,1]$, and $C \geq 1$ there exists a function f of the form $f(x) = \sum_{i=1}^{m} c_i \phi(w_i x + b_i)$, where ϕ is the sigmoid function such that*

$$\forall x \in [0, C], \left| f(x) - \sqrt{x} \right| \leq \epsilon,$$

as long as $m = \Omega\left(\frac{\sqrt{C}}{\epsilon}\right)$.

We can use this lemma along with the result presented in Lemma 11 to get the following corollary:

Corollary 9 (sqrt)**.** *There exists a transformer-based function block with 3 layers and m heads which can implement $f(a) = \sqrt{a}$ up to error $O(1/m)$ for all $a \in [0, O(m^2)]$.*

The functions $f : x \to \frac{1}{x}$ (inversion) and $f : x \to \sqrt{x}$ (square root) since they can be approximated by sums of sigmoids, they can directly be encoded in the standard transformer-based function block form through Lemma 11.

What other functions can our calculator implement? We have included some of the most commonly used operations in calculators in our construction, but it can be extended to include a wider variety of operations such as algebraic and trigonometric functions. When implementing these functions within our transformer architecture, there are typically two choices that can be made. One option is to approximate the target function $f(x)$ using sigmoids. Another option is to use an iterative numerical algorithm where the next output y is calculated based on the previous output y and the goal is to minimize the difference between the calculated output and the target function $f(x)$. This algorithm takes the form $y_{k+1} = g(y_k)$, where g is typically an algebraic function. The desired accuracy is achieved when the difference between the calculated output and target function is less than or equal to a certain tolerance ϵ.

N Linear Algebra

In Appendix J, we demonstrated the implementation of matrix transpose and matrix multiplication as transformer-based function blocks. Utilizing these implementations, we proceed to execute two iterative algorithms for determining the inverse of a matrix through the Newton-Raphson Method and identifying the eigenvector corresponding to the maximum eigenvalue through the Power Iteration method.

Linear algebra using Transformers In the study conducted by Charton (2021), the author implemented some standard matrix method operations using a transformer-based architecture. Four distinct encoding schemes were proposed and applied to nine different operations, ranging from matrix multiplication to eigenvalue decomposition. We find that the size of the networks in Charton (2021) is comparable to that of ours.

As an example we illustrate a comparison of the sizes for matrix transposition in Appendix N. Notice that the number of layers, heads and width may seem different in Appendix N and Lemma 13; however, in the proof of Lemma 13 we first vectorize the matrix (1 layer), then we implement the fixed permutation using Lemma 4 (1 layer) and finally we use another 2 layers to bring back the matrix in its original representation. If the matrix is given to us, as in Charton (2021), in its transposed form then we only need one layer and the two sets of encodings to perform the fixed permutation. Since the maximum size of the matrix is 30×30, the sequence length is $n = 30^2$ and thus the size of each of the encodings will be 10, leading to an input with width $2 \cdot 10 + 1 = 21$. This will lead to a total width of 42, due to the ReLU layer in Lemma 4 having a width double the input's width.

We intend to further investigate our constructions, by implementing them and evaluating the errors involved as a function of the constants used in the proof of Lemma 14 and the temperature in Lemma 3, in future work.

	Layers	Heads	Width
Ours	1	1	42
Charton (2021)	1	8	256

Table 1: Comparison for transposing any matrix of size 30×30 stored as a vector.

Matrix Inversion. We can use the Unified Attention Based Computer to write a program for Matrix Inversion using the functions for matrix multiplications and a function for subtraction. We do so by implementing Newton's algorithm for matrix inversion using our unified framework. The pseudo code for the algorithm is as follows:

Algorithm 8 Pseudocode for running Newton's algorithm for Matrix inversion for T iterations.

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

Lemma 21. *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. 8 with output $\mathbf{X}_1^{(\text{transf})}$ that satisfies $\|\mathbf{X}_1^{(\text{transf})} - \mathbf{X}_1\| \leq \epsilon$. This error ϵ arises due to softmax, and can be driven arbitrarily close to 0 by increasing the temperature.*

Proof. The proof of this lemma is the code using the `FLEQ` instruction provided below (Alg. 9). Let f_{mul}, f_{sub} and f_{transp} be the functions that implement multiplication, substraction and transpose respectively. Then, the following code runs Newton's algorithm for matrix inversion.

Algorithm 9 Program to compute the approximate inverse using our Unified Attention Based Computer

```
Require: $\texttt{mem}[a] = \mathbf{A}$.                                    {This is the location of the input.}
Require: $\texttt{mem}[p] = 2\mathbf{I}, \texttt{mem}[x] = \epsilon\mathbf{I}, \texttt{mem}[y] = \mathbf{0}, \texttt{mem}[q] = -1$.            {Constants.}
Require: $\texttt{mem}[t] = -T$.                             {Iteration counter, $i$ initialized as $i := -T$.}

1: $\texttt{mem}[x] = f_{\text{mul}}(\texttt{mem}[x], \texttt{mem}[a])$.                 {Initializes the result, $\mathbf{X}_{-T} := \epsilon\mathbf{A}$.}
2: $\texttt{mem}[a] = f_{\text{transp}}(\texttt{mem}[a], \texttt{mem}[y])$                                  {Transpose $\mathbf{A}$.}
3: $\texttt{mem}[y] = f_{\text{mul}}(\texttt{mem}[a], \texttt{mem}[x])$.        {First sub-step of Newton's algorithm, $\mathbf{Y} := \mathbf{A}\mathbf{X}_i$}
4: $\texttt{mem}[y] = f_{\text{sub}}(\texttt{mem}[p], \texttt{mem}[y])$.    {Second sub-step of Newton's algorithm, $\mathbf{Y} := 2\mathbf{I} - \mathbf{Y}$}
5: $\texttt{mem}[y] = f_{\text{transp}}(\texttt{mem}[y], \texttt{mem}[q])$.                             {Transpose of $\mathbf{Y}$.}
6: $\texttt{mem}[x] = f_{\text{mul}}(\texttt{mem}[x], \texttt{mem}[y])$.                {Updating the result, $\mathbf{X}_{i+1} := \mathbf{X}_i\mathbf{Y}$}
7: $\texttt{mem}[t] = f_{\text{sub}}(\texttt{mem}[t], \texttt{mem}[q])$.                      {Increment counter, $i := i + 1$.}
8: if $\texttt{mem}[t] \leq 0$ goto instruction 3.                {Keep looping back as long as $i \leq 0$.}
9: EOF.                                                          {End of File command.}
```

Power Iteration. The Power Iteration algorithm (Alg. 10) is used for finding the dominant eigenvalue, the one that has the maximum absolute value, and corresponding eigenvector of a diagonalizable matrix. The algorithm starts with an initial approximation of the eigenvector and converges linearly to the eigenvector associated with the dominant eigenvalue; below we provide the pseudocode.

Algorithm 10 Power Iteration

```
Input: $\mathbf{A}, T$
Initialize $b_0 = \mathbf{1}$
for $k = 0, \dots, T-1$ do
    $\mathbf{b}_{k+1} = \mathbf{A}\mathbf{b}_k$
end for
$\mathbf{b} = \frac{\mathbf{b}_T}{\|\mathbf{b}_T\|}$
```

The last step in the algorithm above needs a normalization by the norm of b_T. While we can compute $\|b_T\|^2$ easily and precisely using the matrix multiplication function block (since $\|b_T\|^2 = b_T^\top b_T$), computing the norm and taking its inverse using the function block from Appendix M would induce error. Hence, we use the following Newton's algorithm that converges quadratically.

Algorithm 11 Newton's algorithm to compute inverse square root: $1/\sqrt{S}$

```
Input: $S$
Initialize $x_0 = 1$
for $k = 0, \ldots, T$ do
    $x_{k+1} = x_k \left(\frac{3}{2} - \frac{S}{2} x_k^2\right)$
end for
```

Lemma 22. *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. 10 for $T = O(\log 1/\epsilon)$ iterations with output $\mathbf{b}_{T+1}^{(\textit{transf})}$ that satisfies $\|\mathbf{b}_{T+1}^{(\textit{transf})} - \mathbf{b}_{T+1}\| \leq \epsilon$. This error ϵ arises due to softmax, and can be driven arbitrarily close to 0 by increasing the temperature.*

Proof. The proof consists of translating each step of the pseudocode for Alg. 10 and Alg. 11 to commands of our unified framework.

Algorithm 12 Program to simulate Power Iteration using our Unified Attention Based Computer

```
Require: $\texttt{mem}[a] = \mathbf{A}, \texttt{mem}[b] = \mathbf{1}, \texttt{mem}[\text{inv\_norm}] = 1.$   {Location of matrix and initialization.}
Require: $\texttt{mem}[q] = 1, \texttt{mem}[p] = 0, \texttt{mem}[r] = 0.5, \texttt{mem}[s] = 1.5$   {Constants.}
Require: $\texttt{mem}[t_1] = \texttt{mem}[t_2] = -T + 1,$

 1: $\texttt{mem}[a] = f_{\text{transp}}(\texttt{mem}[a], \texttt{mem}[p]).$   {Transpose of $\mathbf{A}$.}
 2: $\texttt{mem}[b] = f_{\text{mul}}(\texttt{mem}[a], \texttt{mem}[b]).$   {Inner product: $\mathbf{A}\mathbf{b}_k$.}
 3: $\texttt{mem}[t] = f_{\text{add}}(\texttt{mem}[t_1], \texttt{mem}[q]).$   {Increment counter, $i := i + 1$.}
 4: if $\texttt{mem}[t_1] \leq 0$ goto instruction 2.   {Keep looping back as long as $i \leq 0$.}
 5: $\texttt{mem}[\text{norm\_square}] = f_{\text{mul}}(\texttt{mem}[b], \texttt{mem}[b]).$   {Calculate $\|\mathbf{b}_T\|^2$.}
    Code for Alg. 11 begins.
 6: $\texttt{mem}[\text{y}] = f_{\text{mul}}(\texttt{mem}[\text{inv\_norm}], \texttt{mem}[\text{inv\_norm}]).$   {Calculate $x_k^2$.}
 7: $\texttt{mem}[\text{y}] = f_{\text{mul}}(\texttt{mem}[\text{norm\_square}], \texttt{mem}[\text{y}]).$   {Calculate $Sx_k^2$.}
 8: $\texttt{mem}[\text{y}] = f_{\text{mul}}(\texttt{mem}[r], \texttt{mem}[\text{y}]).$   {Calculate $Sx_k^2/2$.}
 9: $\texttt{mem}[\text{y}] = f_{\text{sub}}(\texttt{mem}[s], \texttt{mem}[\text{y}]).$   {Calculate $(3 - Sx_k^2)/2$.}
10: $\texttt{mem}[\text{inv\_norm}] = f_{\text{mul}}(\texttt{mem}[\text{inv\_norm}], \texttt{mem}[\text{y}]).$   {Update $x_{k+1} := x_k(3 - Sx_k^2)/2$.}
11: $\texttt{mem}[t_2] = f_{\text{add}}(\texttt{mem}[t_2], \texttt{mem}[q]).$   {Increment counter, $j := j + 1$.}
12: if $\texttt{mem}[t_2] \leq 0$ goto instruction 6.   {Keep looping back as long as $j \leq 0$.}
    Code for Alg. 11 ends.
13: $\texttt{mem}[b] = f_{\text{mul}}(\texttt{mem}[b], \texttt{mem}[\text{inv\_norm}]).$   {$\mathbf{b} := \mathbf{b}_T/\|\mathbf{b}_T\|$.}
14: EOF.   {End of File command.}
```

What other numerical linear algebra algorithms can transformers implement? The algorithms presented above serve as proof of concept for the potential to build small linear algebra libraries using our transformer construction. As demonstrated, the size of the looped transformer is constant regardless of the depth. To implement iterative numerical algorithms, additional functions can be incorporated into our architecture. For instance, QR decomposition, Gauss-Seidel, Arnoldi iteration, or Lanczos algorithm can be implemented. While we have not included detailed code for these specific algorithms, the above examples should provide sufficient insight on how to do so.

O Emulating Learning Algorithms at Inference Time

In this section we demonstrate the ability of our unified template to emulate Stochastic Gradient Descent (SGD). We begin by examining the case of linear models, before progressing to the implementation of the backpropagation algorithm for two layer neural networks. Utilizing this as a "function" which we call at each step, we demonstrate the application of SGD in updating the implicit weights of a model.

Our work demonstrates that looped transformers can effectively perform in-context learning for a wide range of models and achieve high levels of accuracy, given access to a sufficient number of inference calls/loops. Previous research, such as in Akyürek et al. (2022) and Garg et al. (2022), has limited in-context learning to a single inference call of a deeper transformer model than ours, which restricts the types of models that can be learned and the level of accuracy that can be achieved. To implement complex iterative programs like SGD, either a looped structure transformer or one that grows in size with the program's depth is required, unless widely believed complexity conjectures are falsified. Additionally, this is the first work to show that transformers can implement SGD on more general loss functions and models beyond linear regression.

Stochastic Gradient Descent in linear models. In Alg. 13 we provide the program for running SGD in linear models on mean squared loss. Consider the dataset $\mathcal{D} = \{(\mathbf{x}_1, y_i), \dots, (\mathbf{x}_{|\mathcal{D}|}, y_{|\mathcal{D}|})\}$, where $\mathbf{x}_i$ is the $i-$th data point and y_i its label; and the loss function of the form $\mathcal{L}(\mathbf{w}) = \frac{1}{|\mathcal{D}|}\sum_{i=1}^{|\mathcal{D}|} \|\mathbf{w}^\top \mathbf{x}_i - y_i\|^2$, where $\mathbf{w}$ is the parameter (weight) vector. Hence, the gradient descent takes the form: $\mathbf{w}_{t+1} = \mathbf{w}_t - \eta \sum_{i=1}^{|\mathcal{D}|}(\mathbf{w}^\top \mathbf{x}_i - y_i)\mathbf{x}_i$, where η is the step-size.

The program we present next iterates through the $|\mathcal{D}|$ data points that the user gives and cycles back to the first point after one pass is completed. The step-size is given as input by the user.

Lemma 23. *There exists a transformer with 13 layers, 1 head and dimensionality $O(\log(|\mathcal{D}|) + d)$ that uses the Unified Attention Based Computer framework in Appendix C.2 to simulate T iterations of SGD on a weight vector $\mathbf{w} \in \mathbb{R}^d$, over a set of $|\mathcal{D}|$ data points $(\mathbf{x}_i, y_i) \in \mathbb{R}^{d+1}$, $i = 1, \dots, |\mathcal{D}|$. The step size is given as a parameter to the program. The simulation 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.*

Remark 3. *The error here that we mention in the theorem statement comes from the matrix multiplication, read, and write operations. The softmax temperature can be used to drive the error of read and write arbitrarily close to 0, while the multiplication also has another free parameter (see* (11)*) that can be used to drive the error of matrix multiplication arbitrarily close to 0, while not affecting the transformer size or architecture.*

Algorithm 13 Program to simulate SGD on linear model under mean squared loss using our Unified Attention Based Computer

```
Require: $\mathtt{mem}[w] = \mathbf{w}, \mathtt{mem}[\eta] = \eta$.   {Location of the weight and step-size.}
Require: $\mathtt{mem}[x_0 + i - 1] = \mathbf{x}_i, i = 1, \ldots, |\mathcal{D}|$.   {Location of the data points.}
Require: $\mathtt{mem}[y_0 + i - 1] = y_i, i = 1, \ldots, |\mathcal{D}|$.   {Location of the labels.}
Require: $\mathbf{p}_{x_*} = x_0$.   {$\mathbf{p}_{x_*}$ is a pointer to the first data. }
Require: $\mathbf{p}_{y_*} = y_0$.   {$\mathbf{p}_{y_*}$ is a pointer to the first label. }
Require: $\mathbf{p}_{\mathtt{PC}} = \text{instr}_1$.   {Program Counter points to first instruction. }
Require: $\mathtt{mem}[q] = 1, \mathtt{mem}[p] = 0, \mathtt{mem}[z] = n$.   {Constants.}
Require: $\mathtt{mem}[j] = -n_d$.   {Within epoch iteration counter initialized to $-n$.}
Require: $\mathtt{mem}[k] = -T$.   {Epoch counter initialized to $-T$.}
1: ( $\text{instr}_1$)  $\mathtt{mem}[temp] = f_{\text{mul}}(\mathtt{mem}[\mathbf{p}_{x_*}], \mathtt{mem}[w])$.   {Inner product: $\mathbf{w}^\top \mathbf{x}_i$.}
2: ( $\text{instr}_2$)  $\mathtt{mem}[temp] = f_{\text{sub}}(\mathtt{mem}[temp], \mathtt{mem}[\mathbf{p}_{y_*}])$.   {Substract the label: $\mathbf{w}^\top \mathbf{x}_i - y_i$.}
3: ( $\text{instr}_3$)  $\mathtt{mem}[temp] = f_{\text{mul}}(\mathtt{mem}[\mathbf{p}_{x_*}], \mathtt{mem}[temp])$.   {Multiply with the data point $\mathbf{x}_i$. }
4: $\mathtt{mem}[temp] = f_{\text{mul}}(\mathtt{mem}[temp], \mathtt{mem}[\eta])$.   {Multiply with the step-size.}
5: $\mathtt{mem}[w] = f_{\text{sub}}(\mathtt{mem}[w], \mathtt{mem}[temp])$.   {Subtract from $\mathbf{w}$ one gradient step.}
6: $\mathtt{mem}[\text{instr}_1] = f_{\text{incr\_pointer}}(\mathtt{mem}[\text{instr}_1])$.   {Increment pointer.}
7: $\mathtt{mem}[\text{instr}_2] = f_{\text{incr\_pointer}}(\mathtt{mem}[\text{instr}_2])$.   {Increment pointer.}
8: $\mathtt{mem}[\text{instr}_3] = f_{\text{incr\_pointer}}(\mathtt{mem}[\text{instr}_3])$.   {Increment pointer.}
9: $\mathtt{mem}[j] = f_{\text{add}}(\mathtt{mem}[j], \mathtt{mem}[q])$.   {Increment within epoch iteration counter by 1.}
10: if $\mathtt{mem}[j] \leq 0$ goto 1.   {Cycle back to the first data point.}
11: $\mathtt{mem}[j] = -n_d$.   {Reset counter.}
12: $\mathtt{mem}[\text{instr}_1] = f_{\text{reset\_pointer}}(\mathtt{mem}[\text{instr}_1], x_0)$.   {Reset pointer.}
13: $\mathtt{mem}[\text{instr}_2] = f_{\text{reset\_pointer}}(\mathtt{mem}[\text{instr}_2], y_0)$.   {Reset pointer.}
14: $\mathtt{mem}[\text{instr}_3] = f_{\text{reset\_pointer}}(\mathtt{mem}[\text{instr}_3], x_0)$.   {Reset pointer.}
15: $\mathtt{mem}[k] = f_{\text{add}}(\mathtt{mem}[k], \mathtt{mem}[q])$.   {Increment epoch counter by 1.}
16: if $\mathtt{mem}[k] \leq 0$ goto 1.   {Cycle back to the first data point.}
17: EOF.   {End of File command.}
```

The following will detail the essential procedures for implementing the Stochastic Gradient Descent algorithm. We employ three pointers, namely $\mathbf{p}_{\mathtt{PC}}$, $\mathbf{p}_{x_*}$ and $\mathbf{p}_{y_*}$ and , in our algorithm. The first one, referred to as program counter, is used to iterate through the commands; after one pass over all data points is completed the program counter is reset to the first instruction (line 16), until T full passes have been completed. The second and third ones, referred to as data and label pointer respectively, iterate through the data points and labels one by one. The increment of the pointer $\mathbf{p}_{x_*}$ needs to occur in both instructions 1 and 3, as to in the next iteration they have been updated from $\text{instr}_i(\mathbf{p}_{x_*}, w, temp) \to \text{instr}_i(\mathbf{p}_{x_*} + 1, w, temp)$, $i = 1, 3$. The same holds for the pointer $\mathbf{p}_{y_*}$ in line 7. Finally, we reset the two pointers in lines 13,14 to cycle back in the first data point, label.

To enhance understanding, we note that lines 6-8 modify the instructions themselves; instead of doing this we could have n_d copies of the lines 1-3, each one with parameters pointers of a different data point,label. In that case the number of *instructions* would have been $7|\mathcal{D}|$.

Notice that the functions $f_{\text{incr\_pointer}}$ and $f_{\text{reset\_pointer}}$ can be directly implemented using Lemma 10.

Note that we can also implement arbitrary loss functions f with updates of the form $\mathbf{w}_{t+1} = \mathbf{w}_t - \eta \sum_{i=1}^{|\mathcal{D}|} f'(\mathbf{w}^\top \mathbf{x}_i - y_i)\mathbf{x}_i$, as long as f' can be well approximated using a Transformer-based Function Block (see Theorem 3 and Lemma 11).

Backpropagation and SGD. We will now generalize the result of Lemma 23 to two layer neural networks with non-linear activation functions; we demonstrate in Alg. 16 how this can be achieved if the activation function is the sigmoid function.

Closest to this section is the work of Akyürek et al. (2022), where the authors prove that constant number of layers is needed to perform one step SGD in linear models, using decoder only transformer architecture.

Algorithm 14 Backpropagation

```
Input: $\mathbf{W}_1 \in \mathbb{R}^{m\times d}$, $\mathbf{b}_1 \in \mathbb{R}^m$, $\mathbf{W}_2 \in \mathbb{R}^{m\times 1}$, $\mathbf{b}_2 \in \mathbb{R}$ , $\mathbf{x} \in \mathbb{R}^d$, $y \in \mathbb{R}$
Compute $\boldsymbol{z} = \mathbf{W}_1\mathbf{x} + \mathbf{b}_1$.
Compute $\boldsymbol{a} = \sigma(\boldsymbol{z})$.
Compute $o = \mathbf{W}_2\boldsymbol{a} + \mathbf{b}_2$.
Compute $\delta_2 = (o - y)$.
Compute $\delta_1 = \sigma'(\boldsymbol{z}) \odot \mathbf{W}_2(o - y)$.
Compute $\frac{\partial J}{\partial \mathbf{W}_2} = \delta_2 \boldsymbol{a}^\top$.
Compute $\frac{\partial J}{\partial \mathbf{b}_2} = \delta_2$.
Compute $\frac{\partial J}{\partial \mathbf{W}_1} = \delta_1 \mathbf{x}^\top$.
Compute $\frac{\partial J}{\partial \mathbf{b}_1} = \delta_1$.
```

Lemma 24. *There exists a transformer with 13 layers, 1 head and dimensionality $O(\log(|\mathcal{D}|) + d)$ that uses the Unified Attention Based Computer framework in Appendix C.2 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 simulation 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.*

Remark 4. *The program we provide in Alg. 15, Specifically, in line 1 of Alg. 16 we call the algorithm for backpropagation at each iteration with a different data point. In terms of our construction, this translates to different instructions which will be in total $O(|\mathcal{D}|)$. As in Alg. 13 the utilization of a pointer that changes the instructions themselves, would result in a program of constant length; we however did not do this to keep the presentation of the algorithm simpler.*

Remark 5. *If we want to account for different activation functions or losses, we can use Lemma 11 to express the gradients as sums of sigmoids. The number of heads (or dimension) would need to be in that case $poly(Tn_d)$ to ensure control over the error induced by the approximation. Another way to achieve low error would be to instead run a Newton's algorithm style algorithm in an inner loop to compute the derivatives (similar to Alg. 12), which will lead to arbitrarily low error without causing the transformer size to increase (at the cost of some extra iterations).*

Algorithm 15 Program to simulate Backpropagation for two layer Neural Networks

```
Input: $\mathbf{p}_{w_1}, \mathbf{p}_{w_2}, \mathbf{p}_{b_1}, \mathbf{p}_{b_2}$    {Pointers to weights and biases.}
Input: $\mathbf{p}_x, \mathbf{p}_y$    {Pointer to data point and label.}
Input: $\eta$.    {Pointer to step size.}
Require: $\mathtt{mem}[q] = 1, \mathtt{mem}[p] = 0, \mathtt{mem}[r] = -1, \mathtt{mem}[m] = m$.    {Constants.}
Require: $\mathtt{mem}[k] = 1$.    {Iteration counter, $k := 1$.}
Require: $\mathbf{p}_z = z^1_{\mathrm{T}}$.    {Pointer for $z$.}
Require: $\mathbf{p}_\delta = \delta^1_{1,\mathrm{T}}$.    {Pointer for $\delta_1$.}
 1: ( instr$_1$) $\mathtt{mem}[temp] = f_{\text{trans}}(\mathtt{mem}[\mathbf{p}_{w_1}], \mathtt{mem}[p])$.    {Create $\mathbf{W}_1^\top$.}
 2: $\mathtt{mem}[z] = f_{\text{mul}}(\mathtt{mem}[temp], \mathtt{mem}[\mathbf{p}_x])$.    {Multiply: $\mathbf{W}_1\mathbf{x}$.}
 3: $\mathtt{mem}[z] = f_{\text{add}}(\mathtt{mem}[z], \mathtt{mem}[\mathbf{p}_{b_1}])$.    {Add the bias: Compute $\boldsymbol{z}$. }
 4: $\mathtt{mem}[a] = f_{\text{sigmoids}}(\mathtt{mem}[z], \mathtt{mem}[q])$.    {Compute $\boldsymbol{a} = \sigma(\boldsymbol{z})$. }
 5: $\mathtt{mem}[temp] = f_{\text{trans}}(\mathtt{mem}[\mathbf{p}_{w_2}], \mathtt{mem}[p])$.    {Create $\mathbf{W}_2^\top$.}
 6: $\mathtt{mem}[o] = f_{\text{mul}}(\mathtt{mem}[temp], \mathtt{mem}[a])$.    {Multiply: $\mathbf{W}_2\boldsymbol{a}$.}
 7: $\mathtt{mem}[o] = f_{\text{add}}(\mathtt{mem}[o], \mathtt{mem}[\mathbf{p}_{b_2}])$.    {Add bias: Compute $o$.}
 8: $\mathtt{mem}[\delta_2] = f_{\text{sub}}(\mathtt{mem}[o], \mathtt{mem}[\mathbf{p}_y])$.    {Compute $\delta_2$.}
 9: $\mathtt{mem}[\delta_1] = f_{\text{mul}}(\mathtt{mem}[\mathbf{p}_{w_2}], \mathtt{mem}[\delta_2])$.    {Multiply $\mathbf{W}_2\delta_2$.}
10: $\mathtt{mem}[flag] = f_{\text{sub}}(\mathtt{mem}[k], \mathtt{mem}[m])$.    {Create $k - m$.}
11: $\mathtt{mem}[\mathbf{p}_z] = f_{\text{trans}}(\mathtt{mem}[z], \mathtt{mem}[p])$.    {Store $\boldsymbol{z}$ to consecutive memory cells.}
12: $\mathtt{mem}[\mathbf{p}_\delta] = f_{\text{trans}}(\mathtt{mem}[\delta_1], \mathtt{mem}[p])$.    {Store $\delta_1$ to consecutive memory cells.}
13: if $\mathtt{mem}[flag] \leq 0$ goto 20.    {If we iterated all the elements goto next command. }
14: ( instr$_{14}$) $\mathtt{mem}[temp'] = f_{\text{sigmoids}}(\mathtt{mem}[p], \mathtt{mem}[\mathbf{p}_z])$.    {Create $\sigma(z_i)$.}
15: $\mathtt{mem}[temp''] = f_{\text{sub}}(\mathtt{mem}[q], \mathtt{mem}[temp'])$.    {Create $1 - \sigma(z_i)$.}
16: $\mathtt{mem}[temp'] = f_{\text{mul}}(\mathtt{mem}[temp'], \mathtt{mem}[temp''])$.    {Create $\sigma'(z_i) = \sigma(z_i)(1 - \sigma(z_i))$.}
17: ( instr$_{17}$) $\mathtt{mem}[\mathbf{p}_\delta] = f_{\text{mul}}(\mathtt{mem}[temp'], \mathtt{mem}[\mathbf{p}_\delta])$.    {Create $\sigma'(z_i)(\mathbf{W}_2)_i(o - y)$.}
18: $\mathtt{mem}[\text{instr}_{14}] = f_{\text{incr\_pointer}}(\mathtt{mem}[\text{instr}_{14}])$.    {Point to next element of $z$.}
19: $\mathtt{mem}[\text{instr}_{17}] = f_{\text{incr\_pointer}}(\mathtt{mem}[\text{instr}_{17}])$.    {Point to next element of $\delta_1$.}
20: $\mathtt{mem}[k] = f_{\text{add}}(\mathtt{mem}[k], \mathtt{mem}[q])$.    {Increment counter, $k := k + 1$.}
21: If $\mathtt{mem}[p] \leq 0$ goto 13.    {Loop back.}
22: $\mathtt{mem}[\text{instr}_1] = f_{\text{reset\_pointer}}(\mathtt{mem}[\text{instr}_{14}], z^1_\top)$.    {Reset pointer.}
23: $\mathtt{mem}[\text{instr}_{15}] = f_{\text{reset\_pointer}}(\mathtt{mem}[\text{instr}_{15}], \delta^1_{1,\top})$.    {Reset pointer.}
24: $\mathtt{mem}[grad\_W_2] = f_{\text{mul}}(\mathtt{mem}[\delta_2], \mathtt{mem}[a])$.    {Create $\frac{\partial J}{\partial \mathbf{W}_2}$.}
25: $\mathtt{mem}[grad\_b_2] = f_{\text{mul}}(\mathtt{mem}[\delta_2], \mathtt{mem}[q])$.    {Create $\frac{\partial J}{\partial \mathbf{b}_2}$.}
26: $\mathtt{mem}[grad\_W_1] = f_{\text{mul}}(\mathtt{mem}[\delta_1], \mathtt{mem}[\mathbf{p}_x])$.    {Create $\frac{\partial J}{\partial \mathbf{W}_1}$.}
27: $\mathtt{mem}[grad\_b_1] = f_{\text{mul}}(\mathtt{mem}[\delta_1], \mathtt{mem}[q])$.    {Create $\frac{\partial J}{\partial \mathbf{b}_1}$.}
28: $\mathtt{mem}[temp] = f_{\text{mul}}(\mathtt{mem}[grad_{W_2}], \mathtt{mem}[\eta])$.    {Multiply with step-size.}
29: $\mathtt{mem}[\mathbf{p}_{w_2}] = f_{\text{sub}}(\mathtt{mem}[\mathbf{p}_{w_2}], \mathtt{mem}[temp])$.    {Update $\mathbf{W}_2$.}
30: $\mathtt{mem}[temp] = f_{\text{mul}}(\mathtt{mem}[grad_{W_1}], \mathtt{mem}[\eta])$.    {Multiply with step-size.}
31: $\mathtt{mem}[\mathbf{p}_{w_1}] = f_{\text{sub}}(\mathtt{mem}[\mathbf{p}_{w_1}], \mathtt{mem}[temp])$.    {Update $\mathbf{W}_1$.}
32: $\mathtt{mem}[temp] = f_{\text{mul}}(\mathtt{mem}[grad_{b_2}], \mathtt{mem}[\eta])$.    {Multiply with step-size.}
33: $\mathtt{mem}[\mathbf{p}_{b_2}] = f_{\text{sub}}(\mathtt{mem}[\mathbf{p}_{b_2}], \mathtt{mem}[temp])$.    {Update $\mathbf{b}_2$.}
34: $\mathtt{mem}[temp] = f_{\text{mul}}(\mathtt{mem}[grad_{b_1}], \mathtt{mem}[\eta])$.    {Multiply with step-size.}
35: $\mathtt{mem}[\mathbf{p}_{b_1}] = f_{\text{sub}}(\mathtt{mem}[\mathbf{p}_{b_1}], \mathtt{mem}[temp])$.    {Update $\mathbf{b}_1$.}
```

Algorithm 16 Program to simulate SGD using our Unified Attention Based Computer

```
Require: $\texttt{mem}[w_1] = \mathbf{W}_1, \texttt{mem}[w_2] = \mathbf{W}_2$.    {Location weights and biases.}
Require: $\texttt{mem}[b_1] = \mathbf{b}_1, \texttt{mem}[b_2] = \mathbf{b}_2$.    {Location of biases.}
Require: $\texttt{mem}[x_0 + i - 1] = \mathbf{x}_i, i = 1, \dots, n_d$.    {Location of the data points.}
Require: $\texttt{mem}[y_0 + i - 1] = y_i, i = 1, \dots, n_d$.    {Location of the labels.}
Require: $\texttt{mem}[z] = \boldsymbol{e}$.    {Indicator for the choice of loss function}
Require: $\mathbf{p}_{x_*} = x_0$.    {$\mathbf{p}_{x_*}$ is a pointer to the first data. }
Require: $\mathbf{p}_{y_*} = y_0$.    {$\mathbf{p}_{y_*}$ is a pointer to the first label. }
Require: $\mathbf{p}_{\texttt{PC}} = \text{instr}_1$.    {Program Counter points to first instruction. }
Require: $\texttt{mem}[q] = 1, \texttt{mem}[p] = 0, \texttt{mem}[z] = n$.    {Constants.}
Require: $\texttt{mem}[j] = -n_d$.    {Within epoch iteration counter initialized to $-n$.}
Require: $\texttt{mem}[k] = -T$.    {Epoch counter initialized to $-T$.}
 1: Backpropagation$(w_1, w_2, b_1, b_2, \mathbf{p}_{x_*}, \mathbf{p}_{y_*})$ {Perform one step of SGD using Backpropagation}
 2: $\texttt{mem}[j] = f_{\text{add}}(\texttt{mem}[j], \texttt{mem}[q])$.    {Increment within epoch iteration counter by 1.}
 3: $\mathbf{p}_{x_*} = f_{\text{incr\_pointer}}(\mathbf{p}_{x_*})$.    {Show to next data point.}
 4: $\mathbf{p}_{y_*} = f_{\text{incr\_pointer}}(\mathbf{p}_{y_*})$    {Show to next label.}
 5: if $\texttt{mem}[j] \leq 0$ goto 1.    {Cycle back until all data points are iterated.}
 6: $\texttt{mem}[j] = -n_d$.    {Reset counter.}
 7: $\mathbf{p}_{x_*} = f_{\text{reset\_pointer}}(\mathbf{p}_{x_*}, x_0)$.    {Reset pointer.}
 8: $\mathbf{p}_{y_*} = f_{\text{reset\_pointer}}(\mathbf{p}_{y_*}, y_0)$.    {Reset pointer.}
 9: $\texttt{mem}[\text{instr}_3] = f_{\text{reset\_pointer}}(\texttt{mem}[\text{instr}_3], x_0)$.    {Reset pointer.}
10: $\texttt{mem}[k] = f_{\text{add}}(\texttt{mem}[k], \texttt{mem}[q])$.    {Increment epoch counter by 1.}
11: if $\texttt{mem}[k] \leq 0$ goto 1.    {Cycle back to the first data point.}
12: EOF.    {End of File command.}
```