Gottfried Wilhelm Leibniz Universität Hannover
Institut für Theoretische Informatik

Theoretical Foundations of Transformer Networks

Master's Thesis

Kai Christian Hallmann
Matriculation Number: 10019681

Hannover, 2024-04-26

First Examiner: Prof. Dr. rer. nat. Heribert Vollmer
Second Examiner: PD Dr. rer. nat. habil. Arne Meier
Advisor: Laura Strieker, M. Sc.

Erklärung der Selbstständigkeit

Hiermit versichere ich, die vorliegende Masterarbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet zu haben. Die Arbeit hat in gleicher oder ähnlicher Form noch keinem anderen Prüfungsamt vorgelegen.

Hannover, 2024-04-26

Kai Christian Hallmann

Contents

1 Introduction

Transformer networks are a type of deep neural network that was initially introduced in 2017 for sequence modeling and transduction, such as language modeling and machine translation. They have been very successful in these tasks and have brought AI-based tools such as OpenAI's ChatGPT ("Chat Generative Pre-trained Transformer") into widespread use. What sets them apart from previously established deep neural networks like long short-term memory and gated recurrent neural networks is that they employ a mechanism called multi-head attention to be more parallelizable. This results in faster training times compared to these previous network architectures [VSP+17].

In this work, we give an introduction to how transformer networks are constructed and how they work. We then continue by viewing them from a more theoretical point of view as recognizers of formal languages, in order to compare them to known complexity classes and reach a better understanding of how powerful they are. In the process, we present many results and give an overview of how they relate to each other. We will also see that these results vary drastically based on some assumptions we make in our theoretical model of transformer networks. However, with the most reasonable assumptions, we will establish the circuit complexity class log-uniform TC^0 as an upper bound.

Such an upper bound helps answer the question, which problems transformer networks are fundamentally unable to solve, at least for sufficiently large inputs. Knowing this might help decide when to switch to a different approach, rather than to spend more time and computational power on training a transformer network only to find out that it is still unable to preform the desired task. We will also consider whether it is because of their high parallelizability, that transformer networks are limited in their capabilities.

2 Transformer Networks

We will begin by giving an overview of the components of transformer networks and how they work, starting from the inside and working our way outward. Next, we will give a precise mathematical description in section 2.3 that will be required in chapter 5. In section 2.4, we will introduce different kinds of attention that we will later use for a theoretical analysis of the complexity of transformer networks. Finally, we will define how transformer networks can be used to recognize formal languages in section 2.5.

The contents of this chapter are largely based on the paper that introduced transformer networks [VSP+17], which we will refer to as their origin.

2.1 Notation

For $n \in \mathbb{N}$, we will write $[n]$ to mean the set of the first n natural numbers $\{m \in \mathbb{N} \mid m < n\} = \{0, 1, \ldots, n-1\}$, $[n]_{+m}$ to mean the set $\{m, m+1, \ldots, m+n-1\}$, and $\log(n)$ to mean the length of the shortest binary string representing n, so $\log(n) = \lceil \log_2(n+1) \rceil$. We will also write $a \equiv_m b$ iff a and b are congruent modulo m.

2.2 Basics

Transformer networks heavily rely on a mechanism called *attention*. Attention was originally designed as an addition to recurrent neural networks for tasks like machine translation. Those recurrent neural networks usually consist of an encoder and a decoder. The encoder reads in a sequence token by token and modifies its internal hidden state at each step based on the token. Then the decoder uses the final hidden state to generate the output tokens.

The problem with this approach is that all information contained in the input sequence has to be compressed into the hidden state, which is a vector of fixed length. As a result, this kind of network can have problems dealing with long input sequences. The attention mechanism allows the decoder to look back at the input tokens that contain the information most relevant to the token that it is about to generate at each step. In other words, the decoder decides which parts of the input sequence to pay attention to. Because of this, the encoder no longer has to encode all information into a

fixed-length vector [BCB15, sections 1-3].

As attention mechanisms allow modeling of dependencies regardless of distance within the input or output sequences, they have become an integral part of recurrent neural networks for tasks such as sequence modeling. Transformer networks, however, forgo the recurrence and instead rely solely on attention to draw global dependencies between input and output. Therefore, they lend themselves more to parallelization. We will begin by describing the attention mechanism they use.

2.2.1 Single Attention

An attention function can be thought of as mapping a set of queries and a set of key-value pairs to an output. These names are chosen to evoke a conceptual similarity to a (series of) lookup(s) in a dictionary data structure. The query is first compared to each key by a similarity function. Then the output is calculated as the sum of the values weighted by the corresponding similarities.

Transformer networks usually utilize scaled dot-product soft attention, so we will begin by describing this type of attention and introduce other types in section 2.4. The query and the keys are d_k-dimensional vectors, and the values and the output are d_v-dimensional vectors. The similarity of the query and a key is evaluated by calculating their dot product and dividing it by $\sqrt{d_k}$. Next, all of these similarities are turned into a probability distribution for each key by applying a function called softmax (see section 2.4) to them. Finally, the output is calculated as the dot product of this distribution and the values.

In practice, this is done for a set of queries at a time because it is more efficient to operate with matrices. We will call the number of queries n_q and the number of key-value pairs n_k. Now the queries are an $n_q \times d_k$ matrix Q, the keys are an $n_k \times d_k$ matrix K, the values are a $n_k \times d_v$ matrix V, and the output is a $n_q \times d_v$ matrix which is calculated as:

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{Q \cdot K^\mathsf{T}}{\sqrt{d_k}}\right) \cdot V$$

The dot products can be calculated very efficiently due to highly optimized code. The scaling factor of $\frac{1}{\sqrt{d_k}}$ is used to limit the size of the values passed into the softmax function to prevent extremely small gradients. Without this scaling, training of a transformer network might come to a halt in the backpropagation phase due to a phenomenon known as the vanishing gradient problem [BJZP20, section 1, subsection 2.1].

2.2.2 Multi-Head Attention

Instead of directly performing single attention on the d_{model}-dimensional vectors that are used by the transformer network internally, they are first split up into shorter vectors. Then the single attention is performed on these shorter vectors in so-called *attention heads*, which can be done in parallel. Finally, the results are put together into a d_{model}-dimensional vector.

Splitting up the vectors is done using learned linear projections. For each of the h heads, these projections are stored as a $d_{\text{model}} \times d_k$ matrix W_i^Q, a $d_{\text{model}} \times d_k$ matrix W_i^K, and a $d_{\text{model}} \times d_v$ matrix W_i^V. The output is assembled by concatenating the outputs of the individual heads and performing another learned linear projection. This one is stored in the $(h \cdot d_v) \times d_{\text{model}}$ matrix W^O. Putting it all together, we get:

$$\begin{aligned}\text{MultiHead}(Q, K, V) &= \text{Concat}(\text{head}_0, \ldots, \text{head}_{h-1}) \cdot W^O \\ \text{head}_i &= \text{Attention}\Big(Q \cdot W_i^Q, K \cdot W_i^K, V \cdot W_i^V\Big) \text{ for } i \in [h]\end{aligned}$$

The values for the dimensions that were originally used are $d_{\text{model}} = 512$, $h = 8$, and $d_k = d_v = \frac{d_{\text{model}}}{h} = 64$.

Splitting the attention between multiple heads has the benefit that each one of them can evaluate different aspects of the queries or view them in a different context.

2.2.3 Position-wise Feed-Forward Networks

Transformer networks also make use of fully connected feed-forward networks. These consist of two linear transformations with a rectified linear unit (ReLU) activation function in between.

$$\text{FFN}(x) = \max(0, x \cdot W_1 + b_1) \cdot W_2 + b_2$$

The input and the output have dimension d_{model}, while the inner layer has dimension $d_{\text{ff}} = 2048$.

Since the input and output dimension is only the size of one position, the feed-forward networks are said to be position-wise. Any interaction between different positions only takes place through the means of attention.

2.2.4 Encoder and Decoder

Transformer networks have an encoder-decoder structure. The encoder is given a sequence of symbol representations $(x_0, \ldots, x_{n-1})$ and maps it to a sequence of continuous representation $\mathbf{z} = (z_0, \ldots, z_{n-1})$. The decoder takes this sequence and outputs an

output sequence $(y_0, \ldots, y_{n-1})$ one element at a time. Transformer networks are autoregressive, meaning that they take the previously generated symbols as an additional input when generating the next symbol.

The encoder is composed of a stack of $L = 6$ identical layers. Each layer consists of two sub-layers. The first one performs multi-head self-attention. This means that it uses the input as the Q, K, and V matrices. The second sub-layer is a simple, position-wise fully connected feed-forward network. The learned parameters of this network differ from layer to layer, whereas the linear transformations are shared between all feed-forward networks. Additionally, each sub-layer contains a residual connection and a layer normalization. That is, the ingoing values are added to the outgoing values and the layers are normalized to be in a desired range, which introduces further learned parameters. All sub-layers in the model produce vectors of size d_{model}.

The decoder is also composed of a stack of $L = 6$ identical layers. It consists of the same two sub-layers as the encoder, along with a third one in between them. In this layer, it once again performs multi-head attention. This time, however, the input to this sub-layer is only used as the Q matrix, while the output $\mathbf{z}$ of the encoder is used as the K and V matrix. The self-attention sub-layer is also slightly modified to prevent positions from attending to subsequent positions. This is done by setting the corresponding values in the matrices to $-\infty$ before applying the softmax function. Just like the encoder, each sub-layer once again contains a residual connection and a layer normalization.

A diagram of the structure of the encoder and decoder of a transformer network is shown in Figure 2.1.

2.2.5 Embedding and Softmax

Additionally, a learned embedding is used to convert the input and output tokens to vectors of dimension d_{model}. Another learned linear transformation followed by a softmax function is used to convert the decoder outputs to the predicted probabilities for the next token. The original authors suggest using the same matrix both for the two embedding layers and for the linear transformation in order to tie the input and output embeddings together, similar to [PW17, section 1]. They also suggest multiplying the weights in the embedding layers by $\sqrt{d_{\text{model}}}$ without motivating this.

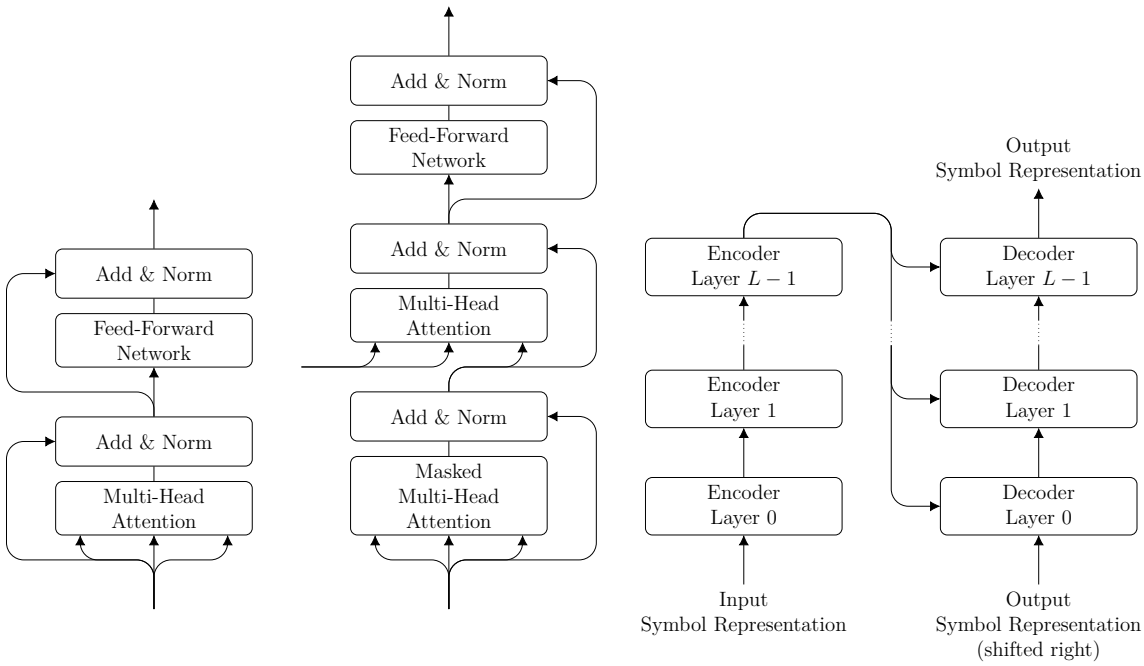

Figure 2.1: An encoder layer (left), a decoder layer (middle), and the complete encoder-decoder structure (right)

2.2.6 Positional Encoding

Transformer networks do not inherently impose any structure on their inputs. Because of this, any such structure, even a linear order, has to be explicitly added to the inputs. The positional encoding suggested by the original authors uses the sine and cosine functions for this:

$$\mathrm{PE}_{(pos,2i)} = \sin\left(\frac{pos}{10000^{\frac{2i}{d_{\text{model}}}}}\right)$$
$$\mathrm{PE}_{(pos,2i+1)} = \cos\left(\frac{pos}{10000^{\frac{2i}{d_{\text{model}}}}}\right)$$

They choose this encoding because it might allow the model to easily learn to attend to relative positions, since for any fixed offset k, PE_{pos+k} can be represented as a linear function of PE_{pos}. This might allow the model to extrapolate to sequence lengths longer than those encountered during training.

A diagram of a full transformer network can be seen in Figure 2.2.

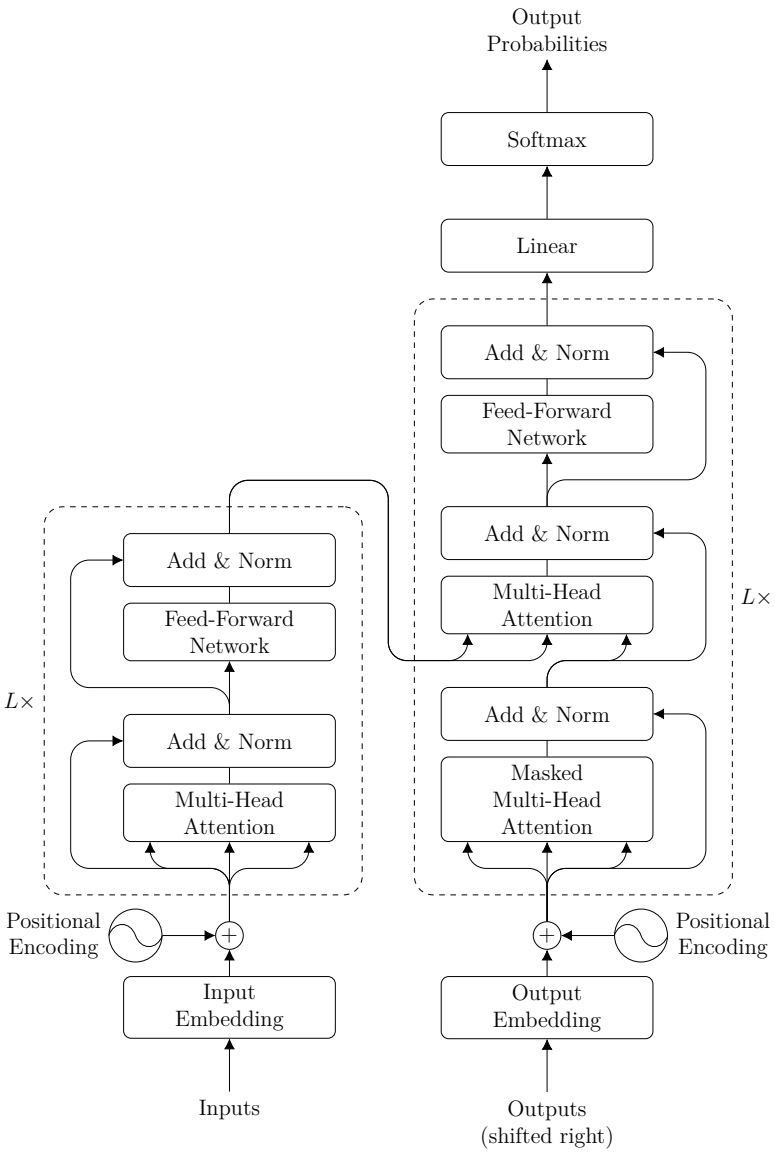

Figure 2.2: The original transformer network

2.2.7 Further Developments in Transformer Network Design

In later transformer networks like Google's BERT and OpenAI's GPT, the structure has been further simplified, getting rid of the encoder-decoder structure in favor of simply a sequence of transformer blocks. Each transformer block is just a layer of the original transformer network's encoder or decoder [DCLT19, section 3; RNSS18, subsections 3.1, 4.1].

Another trend that can be observed is that the number L of transformer blocks/layers of different transformer networks is steadily increasing, as can be seen in Table 2.1. This is done to allow the networks to store ever more information about the larger and larger datasets they are trained on [Ope23].

As most transformer networks are a sequence of transformer blocks together with a more task-specific transformation of the input and output, such as embedding and positional encoding, we will focus on this structure and provide mathematical description that allows a theoretical analysis in the following section 2.3.

Transformer Network	Year	Layers L	Source
original	2017	6	[VSP+17]
GPT-1	2018	12	[RNSS18]
BERT	2018	up to 24	[DCLT19]
GPT-2	2019	up to 48	[RWC+19]
GPT-3	2020	up to 96	[BMR+20]
GPT-4	2023	rumored 120	[Ope23]

Table 2.1: Number of Layers of Different Transformer Networks

2.3 Mathematical Description

Section 2.3, section 2.4, and section 2.5 are based on [MSS22, section 3].

2.3.1 Datatypes

We model all internal data of a transformer network as binary strings. In order to be able to perform calculations with those binary strings, we need a semantics to interpret them as numbers. As is often done in circuit complexity, we will first interpret them as unsigned integers.

Definition 2.1. *A binary string $x \in [2]^*$ of length $|x| = n$ interpreted as an* unsigned integer *has the value*

$$[\![x]\!]_{\mathbb{N}} \coloneqq \sum_{i \in [n]} 2^i \cdot x_{n-1-i}.$$

We denote the standard integer operations on binary strings by $+_{\mathbb{N}}$, $\cdot_{\mathbb{N}}$, and $<_{\mathbb{N}}$.

Additionally, we define fixed-point numbers, which we will only use in section 5.5.

Definition 2.2. *A binary string $x \in [2]^{r+s}$ interpreted as a* fixed-point number *with integer part r, fractional part s, and precision $(r+s)$ has the value*

$$[\![x]\!]_{\mathbb{FP}_{r,s}} \coloneqq \frac{[\![x]\!]_{\mathbb{N}}}{2^s} - \begin{cases} 0, & \text{if } x_{r+s} = 0 \\ 2^r, & \text{otherwise.} \end{cases}$$

That is, x gets interpreted as an integer using the two's complement and that integer is divided by 2^s. Since r and s are fixed, we normally just write $\mathbb{FP}$ in place of $\mathbb{FP}_{r,s}$. Note that division of fixed-point numbers will usually involve rounding of the result and that various over- and underflows can occur which we leave undefined.

Next, we define how we encode rational numbers without a fixed size.

Definition 2.3. *To interpret a binary string $r \in [2]^*$ as a* rational number*, we view it as a sign bit s and an encoded pair of unsigned integer strings $\langle p, q \rangle$. Its numerical value is*

$$[\![r]\!]_{\mathbb{Q}} \coloneqq (-1)^s \cdot \frac{[\![p]\!]_{\mathbb{N}}}{[\![q]\!]_{\mathbb{N}}}.$$

We once again denote the standard operations by $+_{\mathbb{Q}}$ and $\cdot_{\mathbb{Q}}$ and define them to always reduce their result as far as possible.

Finally, we will define *floats* $\mathbb{F}$ as the subset of the rationals $\mathbb{Q}$ where the denominator is constrained to be a power of 2. This resembles the way numbers are usually encoded in most computers more closely than the rational numbers. Addition and multiplication are defined the same way as for $\mathbb{Q}$. Note, however, that the multiplicative inverse of

a float is not necessarily another float and may have to be approximated. Because of this, it may be that $[\![(x/_{\mathbb{F}}y) \cdot_{\mathbb{F}} y]\!]_{\mathbb{F}} \neq [\![x]\!]_{\mathbb{F}}$.

Going forward, we will usually omit datatype subscripts where they are clear from context. We will sometimes write $\mathbb{D}$ as a generic datatype or other datatypes in function signatures to mean $[2]^*$ while making the intent clearer and hinting at the semantics.

We imagine the tuple $\langle p, q \rangle$ to be encoded by padding p, q to the same length using leading 0s and interleaving their bits. This means that the size of a rational number or a float is $2 \cdot \max(|p|, |q|) + 1$. We will use the following property to limit the internal functions in our transformer network model:

Definition 2.4. *We say a function $f\colon [2]^* \to [2]^*$ is* size-preserving *iff there exist constants c, n such that for all inputs x with $|x| \geq n$, the size of the function value is bounded by $|f(x)| \leq c \cdot |x|$. Let $\mathcal{P}$ be the set of size-preserving functions.*

2.3.2 Transformer Networks

Now we give a precise definition of a transformer network that consists of a sequence of transformer blocks rather than an encoder-decoder structure.

Definition 2.5. *A* transformer network *is a tuple*

$$\big(\Sigma, \mathbb{D}, \alpha, L, H, \varphi, (s_{\ell,h})_{\ell \in [L], h \in [H]}, (f_\ell)_{\ell \in [L]}\big)$$

where

1. *Σ is the input alphabet.*
2. *$\mathbb{D}$ is a scalar datatype, that is, a semantics for interpreting binary strings as numbers. We will generally consider $\mathbb{D} = \mathbb{F}$.*
3. *$\alpha\colon \mathbb{D}^* \to \mathbb{D}^*$ is an attention function that maps a vector of attention scores in $\mathbb{D}^n$ to a normalized probability distribution also in $\mathbb{D}^n$. (See section 2.4)*
4. *$L \in \mathbb{N}$ is the number of layers.*
5. *$H \in \mathbb{N}$ is the number of heads.*
6. *$\varphi\colon \Sigma \times \mathbb{N} \to \mathbb{D}^m$ is a position-aware embedding function that maps a token and a position to a vector, where m is a multiple of H.*
7. *For each ℓ, h, the function $s_{\ell,h}\colon \mathbb{D}^m \times \mathbb{D}^m \to \mathbb{D}$ assigns attention scores to pairs of values.*
8. *For each ℓ, the function $f_\ell\colon \mathbb{D}^m \times \mathbb{D}^m \to \mathbb{D}^m$ maps a previous layer output and attention head to a new value vector.*

On an input string $w \in \Sigma^n$, a transformer network computes L layers of output sequences $v_{\ell,0}, \ldots, v_{\ell,n}$ for $\ell \in [L]_{+1}$ where each $v_{\ell,i} \in \mathbb{D}^m$. First, each token w_i and its position i are embedded into a value $v_{0,i}$. Subsequently, each layer ℓ aggregates information from the previous value sequence v_ℓ using a multi-head attention mechanism and outputs a new value sequence $v_{\ell+1}$. The layers are structured as follows:

1. *Embedding layer*: $v_{0,i} = \varphi(w_i, i)$ for $i \in [n]$.

2. *Attention head*: Each of the H attention heads in layer ℓ maps the full previous sequence to a new value via $s_{\ell,h}$ and then applies the attention function α:

$$\begin{aligned} a_{\ell,h,i,j} &= s_{\ell,h}(v_{\ell,i}, v_{\ell,j}) \text{ for } \ell \in [L],\ h \in [H],\ i \in [n],\ j \in [n] \\ b_{\ell,h,i} &= \sum_{j \in [n]} \alpha(a_{\ell,h,i,0}, \ldots, a_{\ell,h,i,n-1})_j \cdot v_{\ell,j} \text{ for } \ell \in [L],\ h \in [H],\ i \in [n]. \end{aligned}$$

   It is important to note that the semantics for addition and multiplication as well as the computation of α come from $\mathbb{D}$.

3. *Activation block*:

$$v_{\ell+1,i} = f_\ell(v_{\ell,i}, (b_{\ell,0,i}, \ldots, b_{\ell,h-1,i})) \text{ for } \ell \in [L],\ i \in [n].$$

   This model combines the aggregation of the attention heads, the feed-forward network, the residual connections and the layer normalizations into the single function f_ℓ. A benefit of this is that it generalizes to changes in the layout like for example moving the layer normalization around, which is one of the changes between GPT-1 and GPT-2 [RWC$^+$19, subsection 2.3].

2.4 Types of Attention

Attention mechanisms make use of an attention function to convert a vector of values $a \in \mathbb{D}^n$ into a probability distribution over $0, 1, \ldots, n-1$. So far, we have used the softmax function for this purpose:

$$\operatorname{softmax}(a)_i = \frac{e^{a_i}}{\sum_{k \in [n]} e^{a_k}}.$$

We will also examine the simpler *hard* and *saturated* attention when evaluating the complexity of transformer networks in chapter 5. In order to define these, we first define the function $\mathcal{M}\colon \mathbb{D}^n \to \mathcal{P}([n])$ that maps a vector of values to the set of indices

of maximum values:

$$\mathcal{M}(a) \coloneqq \{i \mid a_i = \max\{a_j \mid j \in [n]\}\}.$$

Using this, we define *hard attention* (also known as *unique hard attention* [HAF22, subsection 4.2]) to work the same as soft attention but use the hardmax function instead of the softmax function:

$$\text{hardmax}(a)_i \coloneqq \begin{cases} 1, & \text{if } i = \min(\mathcal{M}(a)) \\ 0, & \text{otherwise.} \end{cases}$$

This function returns a one-hot distribution.

Analogously, we define *strong saturated attention* (also known as *averaging hard attention* [HAF22, subsection 4.2]) to use the strongsatmax function instead:

$$\text{strongsatmax}(a)_i \coloneqq \frac{1}{|\mathcal{M}(a)|} \cdot \begin{cases} 1, & \text{if } i \in \mathcal{M}(a) \\ 0, & \text{otherwise.} \end{cases}$$

Finally, we define *weak saturated attention* such that each attention head either uses the hardmax function or the uniformmax function:

$$\text{uniformmax}(a)_i \coloneqq \frac{1}{n}.$$

2.5 Language Recognition

Now we will define language recognition for transformer networks.

Definition 2.6. *Let $v_{\ell,i}(w)$ denote the value of $v_{\ell,i}$ on input string w. A transformer network* recognizes *a formal language $\mathcal{L} \subseteq \Sigma^*$ iff there exists a $\mathbb{D}$-valued affine transformation W, b such that for all $w \in \Sigma^*$ the following holds:*

$$W \cdot v_{L,0}(w) + b > 0 \iff w \in \mathcal{L}.$$

In other words, the decision problem of recognizing $\mathcal{L}$ must be linearly separable using the first value in the last layer of the transformer network.

Finally, we define AHAT($\mathbb{D}$) ("averaging hard attention transformer") as the set of languages recognizable by some saturated transformer network over $\mathbb{D}$ where the internal functions can be any size-preserving functions. To be able to apply the concept of size-preservation to φ, we assume the size of a token to be $\log(|\Sigma|)$.

3 Circuit Complexity

The definitions in this chapter are taken and partially adapted from [Vol99, pp. 7-10, 46-47, 108, 126].

3.1 Circuits and Circuit Families

Definition 3.1. *An n-ary* Boolean function *for some $n \in \mathbb{N}$ is a function $f\colon [2]^m \to [2]$. A* family of Boolean functions *is a sequence $f = (f^n)_{n\in\mathbb{N}}$, where each f^n is an n-ary Boolean function.*

Definition 3.2. *A* basis *is a finite set of Boolean functions and families of Boolean function. The* standard unbounded fan-in basis $B_1 \coloneqq \left\{\neg, (\wedge^n)_{n\in\mathbb{N}}, (\vee^n)_{n\in\mathbb{N}}\right\}$ *contains the unary Boolean* NOT *function and the families of Boolean* AND *and* OR *functions.*

Definition 3.3. *The n-ary Boolean* majority function *checks if at least half of the input bits are* 1*:*

$$\mathrm{MAJ}^n\colon [2]^n \to [2], (x_0, \ldots, x_{n-1}) \mapsto \begin{cases} 1, & |\{i \in [n] \mid x_i = 1\}| \geq \frac{n}{2} \\ 0, & \textit{otherwise} \end{cases}$$

Definition 3.4. *Let B be a basis and $n, m \in \mathbb{N}$. A* Boolean circuit *over B with n inputs and m outputs is a tuple $C = (V, E, \alpha, \beta, \omega)$, where (V, E) is a finite directed acyclic graph, $\alpha\colon E \to \mathbb{N}$ is an injective function, $\beta\colon V \to B \cup \{x_0, \ldots, x_{n-1}\}$, and $\omega\colon V \to \{y_0, \ldots, y_{m-1}\} \cup \{*\}$, such that the following conditions hold:*

1. *If $v \in V$ has in-degree 0, then $\beta(v) \in \{x_0, \ldots, x_{n-1}\}$ or $\beta(v)$ is a 0-ary Boolean function from B.*
2. *If $v \in V$ has in-degree $k > 0$, then $\beta(v)$ is a k-ary Boolean function from B or a family of Boolean functions from B.*
3. *For every $i \in [n]$, there exists at most one node $v \in V$ such that $\beta(v) = x_i$.*
4. *For every $i \in [m]$, there exists exactly one node $v \in V$ such that $\omega(v) = y_i$.*

Definition 3.5. *Let $C = (V, E, \alpha, \beta, \omega)$ be a circuit over B with n inputs and m outputs. First, we inductively define a function $\mathrm{val}_v \colon [2]^* \to [2]$ for every $v \in V$ as follows: Let $a_0, \dots, a_{n-1}$ be arbitrary values.*

1. *If $v \in V$ has fan-in 0 and if $\beta(v) = x_i$ for some $i \in [n]$, then $\mathrm{val}_v(a_0, \dots, a_{n-1}) \coloneqq a_i$. If $v \in V$ has fan-in 0 and if $\beta(v) = b$ is a 0-ary function from B, then $\mathrm{val}_v(a_0, \dots, a_{n-1}) \coloneqq b$.*

2. *Let $v \in V$ have fan-in $k > 0$ and let $v_0, \dots, v_{k-1}$ be the gates that are predecessors of v ordered in such a way that $\alpha((v_1, v)) < \dots < \alpha((v_{k-1}, v))$. Let $\beta(v) = f \in B$. If f is a k-ary function, then let*

$$\mathrm{val}_v(a_0, \dots, a_{n-1}) \coloneqq f(\mathrm{val}_{v_0}(a_0, \dots, a_{n-1}), \dots, \mathrm{val}_{v_{k-1}}(a_0, \dots, a_{n-1})).$$

   Otherwise f must be a family of Boolean functions, $f = (f^n)_{n \in \mathbb{N}}$. In this case, we define

$$\mathrm{val}_v(a_0, \dots, a_{n-1}) \coloneqq f^k(\mathrm{val}_{v_0}(a_0, \dots, a_{n-1}), \dots, \mathrm{val}_{v_{k-1}}(a_0, \dots, a_{n-1})).$$

For $i \in [m]$, let v_i be the unique gate $v_i \in V$ with $\omega(v_i) = y_i$. Then the function computed by *C, $f_C \colon [2]^n \to [2]^m$, is given for all $a_0, \dots, a_{n-1} \in [2]$ by*

$$f_C(a_0, \dots, a_{n-1}) \coloneqq \big(\mathrm{val}_{v_0}(a_0, \dots, a_{n-1}), \dots, \mathrm{val}_{v_{m-1}}(a_0, \dots, a_{n-1})\big).$$

Definition 3.6. *Let B be a basis. A* circuit family *over B is a sequence $\mathcal{C} = (C_0, C_1, \dots)$, where, for every $n \in \mathbb{N}$, C_n is a circuit over B with n inputs. Let f^n be the function computed by C_n. Then we say that $\mathcal{C}$ computes the function $f \colon [2]^* \to [2]^*$, defined for every $w \in [2]^*$ by*

$$f(w) \coloneqq f^{|w|}(w).$$

We write $f = (f^n)_{n \in \mathbb{N}}$ and $\mathcal{C} = (C_n)_{n \in \mathbb{N}}$. We say that $\mathcal{C}$ accepts *$A \subseteq [2]^*$ iff $\mathcal{C}$ computes c_A. In this context, we also use the notation $A = (A^n)_{n \in \mathbb{N}}$ (where $A^n \coloneqq A \cap [2]^n$) and $c_A = (c_{A^n})_{n \in \mathbb{N}}$. If $\mathcal{C}$ is a circuit family, we use the notation $f_{\mathcal{C}}$ for the function computed by $\mathcal{C}$.*

Definition 3.7. *Let $C = (V, E, \alpha, \beta, \omega)$ be a circuit over B. The* size *of C is defined to be the number of non-input gates in V, that is, $|\{v \in V \mid \beta(v) \in B\}|$, and the* depth *of C is defined to be the length of a longest directed path in the graph (V, E).*

Let $\mathcal{C} = (C_n)_{n \in \mathbb{N}}$ be a circuit family and let $s, d \colon \mathbb{N} \to \mathbb{N}$. $\mathcal{C}$ has size *s and* depth *d if, for every n, C_n has size $s(n)$ and depth $d(n)$.*

3.2 Circuit Complexity Classes

For circuit complexity classes

Definition 3.8. *Let B be a basis and let $s, d\colon \mathbb{N} \to \mathbb{N}$. The class* $\mathrm{SIZE\text{-}DEPTH}_B(s, d)$ *contains all sets $A \subseteq [2]^*$ for which there exists a circuit family C over basis B of size $O(s)$ and depth $O(d)$ that accepts A.*

The class of polynomial size constant depth AND/OR circuits is defined as follows:

Definition 3.9. $\mathrm{AC}^0 \coloneqq \mathrm{SIZE\text{-}DEPTH}_{B_1}\left(n^{O(1)}, 1\right)$

It is known to include problems such as integer addition, subtraction, and comparison.

The class of polynomial size constant depth threshold circuits (since it can also be defined in terms of threshold gates rather than majority gates) is defined as follows:

Definition 3.10. $\mathrm{TC}^0 \coloneqq \mathrm{SIZE\text{-}DEPTH}_{B_1 \cup \left\{(\mathrm{MAJ}^n)_{n\in\mathbb{N}}\right\}}\left(n^{O(1)}, 1\right)$

It is known to include problems such as integer multiplication, division, and sorting.

From the definition, it is obvious that AC^0 is included in TC^0. One important result in the field of circuit complexity theory is that this inclusion is a proper one:

Theorem 3.11. $\mathrm{AC}^0 \subsetneq \mathrm{TC}^0$ *[Vol99, Corollary 4.35].*

The proof of this is beyond the scope of this work. As a result of this, it can be shown that, for example, integer multiplication is not included in AC^0.

3.3 Uniformity

One problem with circuit families is that they are infinite objects. The circuits within a given circuit family can be completely different from each other. One of the consequences of this is that undecidable languages, like

$$\bigcup_{\mathrm{bin}(n)\in K} [2]^n,$$

where K is the special halting problem, can be accepted by a circuit family where each circuit just outputs a constant 0 or 1.

Algorithms, however, are finite. A program written in any programming language has a finite text. A Turing machine has a finite number of states and therefore a finite transition function. A random access machine (RAM) has a finite number of instructions. Therefore, a uniform circuit family should be a circuit family with a finite description.

As a finite description of a circuit family $\mathcal{C} = (C_n)_{n\in\mathbb{N}}$, we introduce a function $f_{\mathcal{C}}$, with $1^n \mapsto \langle C_n \rangle$, that is easily computable. In particular:

Definition 3.12. *A circuit family $\mathcal{C} = (C_n)_{n\in\mathbb{N}}$ of size s is* logspace-uniform*, or* log-uniform*, iff there is an admissible encoding scheme such that the function $f_{\mathcal{C}}, 1^n \mapsto \langle C_n \rangle$ is in* FDSPACE($\log(s)$).

4 Logic

In this chapter, we introduce two extensions of first order logic that are used to define formal languages. They use variables indexing the symbols of a given word to describe characteristics that define the language. We will use these logics in chapter 5 to bound the complexity of transformer networks.

4.1 First Order Logic with Counting

This section is based on [CCP23, section 4].

We will describe the syntax of FOC[+; MOD], given a fixed (finite) alphabet Σ, and its intended interpretation with reference to (finite) strings $w \in \Sigma^n$ for some $n \in \mathbb{N}$. It consists of

- *position variables* $p, \ldots$ which stand for positions in w, that is integers in $[n]$,
- *count variables* $x, y, z, \ldots$ which stand for rational numbers,
- *(count) terms* $c_0 \cdot x_0 + \cdots + c_{k-1} \cdot x_{k-1} + c_k$, where each c_i is a rational number and each x_i is a count variable.

A formula of FOC[+; MOD] is one of:

- $\top$ for true and $\bot$ for false.
- $Q_a(p)$ where $a \in \Sigma$, which is true iff $w_p = a$.
- $\mathrm{MOD}_m^r(p)$ where $r \geq 0$, $m > 0$, which is true iff $p \equiv_m r$.
- $t_0 = t_1$, $t_0 < t_1$ where t_0 and t_1 are terms, which follow the conventional semantics.
- $\varphi_0 \wedge \varphi_1$, $\varphi_0 \vee \varphi_1$, $\neg\varphi_0$ where φ_0 and φ_1 are formulae, which follow the conventional semantics.
- $\exists x.\varphi$, $\forall x.\varphi$ where x is a count variable and φ is a formula, which follow the conventional semantics.
- $\exists^{=x} p.\varphi$, where x is a count variable, p is a position variable, and φ is a formula, which is true iff φ is true for exactly x values of p. (Note that $\exists^{=x} p.\varphi$ only binds p.)

We will use the following abbreviations:

- $\varphi \to \psi \coloneqq \neg\varphi \vee \psi$
- $\varphi \leftrightarrow \psi \coloneqq (\varphi \to \psi) \wedge (\psi \to \varphi)$
- $\exists p.\varphi \coloneqq \exists x.(x > 0 \wedge \exists^{=x} p.\varphi)$
- $\forall p.\varphi \coloneqq \exists x.(\exists^{=x} p.\top \wedge \exists^{=x} p.\varphi)$

We call any variable that is not bound by a quantifier *free*, and a formula with no free variables a *sentence*. For a sentence σ and a string $w \in \Sigma^*$, we write $w \models \sigma$ iff w makes σ true.

Definition 4.1. *If σ is a sentence of* FOC[+; MOD]*, the language defined by σ is* $L(\sigma) \coloneqq \{w \mid w \models \sigma\}$.

The part of the logic that deals with position variables is like monadic first-order logic, in which all predicates are monadic (that is, they take only one argument). The other part of the logic that deals with count variables is the theory of rational numbers with ordering and addition (but not multiplication). Both of these other logics have useful normal forms: Monadic first-order logic has a normal form that uses only one variable, while the theory of rationals with ordering and addition has quantifier elimination. We can combine these two results to get a very simple normal form for FOC[+; MOD].

Theorem 4.2. *Every formula φ of* FOC[+; MOD] *is equivalent to a formula of the form*

$$\varphi' = \exists x_0 \ldots \exists x_{k-1}. \left(\bigwedge_i \exists^{=x_i} p.\psi_i \wedge \chi \right)$$

where each ψ_i is quantifier-free and has no free count variables and χ is quantifier-free.

Proof. See [CCP23, Appendix A]. □

It may seem odd that count variables range over rational numbers, when counts are always integers. This technicality simplifies the normal form: If we had used integers, then the part of the logic that deals with count variables would be Presburger arithmetic, and the normal form would require allowing $\mathrm{MOD}^r_m(x)$ on count variables as well.

4.2 First Order Logic with Majority

This section is based on [MS23a, subsection 2.2].

We will describe the syntax of FO(M), given a fixed (finite) alphabet Σ, and its intended interpretation with reference to (finite) strings $w \in \Sigma^n$ for some $n \in \mathbb{N}$. A term is

- a constant 0, 1, or $n-1$,
- an *(index) variable* $i, j, k, \ldots$ ranging from 0 to $n-1$,
- any sum $t_0 + t_1$ or difference $t_0 - t_1$ of terms t_0 and t_1.

A formula is one of:

- $Q_a(t)$ where $a \in \Sigma$ and t is a term, which is true iff $w_t = a$.
- $\mathrm{bit}(\mathrm{t}_0, \mathrm{t}_1)$ where t_0 and t_1 are terms, which is true iff the t_1-th bit of the binary expansion of the value of t_0 is a 1.
- $t_0 = t_1$, $t_0 \leq t_1$, $t_0 \geq t_1$ where t_0 and t_1 are terms, which follow the conventional semantics.
- $\varphi_0 \wedge \varphi_1$, $\varphi_0 \vee \varphi_1$ where φ_0 and φ_1 are formulae, which follow the conventional semantics.
- $\exists i.\varphi$, $\forall i.\varphi$ where i is an index variable and φ is a formula, which follow the conventional semantics.
- $\mathrm{M}x.\varphi$ where i is an index variable and φ is a formula, which is true iff $\geq \frac{n}{2}$ values of i make φ true.

Just like with FOC[+; MOD], we call a formula with no free variables a *sentence*. For a sentence σ and a string $w \in \Sigma^*$, we write $w \models \sigma$ iff w makes σ true.

Definition 4.3. *If σ is a sentence of* FO(M)*, the language defined by σ is $L(\sigma) \coloneqq \{w \mid w \models \sigma\}$.*

Beyond this, FO(M) can express *counting* and *threshold* quantifiers in terms of majority quantifiers. Given a formula φ, a counting quantifier $\exists^k$ creates a new formula $\exists^k i.\varphi$ that is true iff φ is true across exactly k values of i. Threshold quantifiers $\exists^{\leq k}$ and $\exists^{\geq k}$ work similarly but check if φ is true for at least or at most k values of i. In addition, FO(M) can express conditional majority quantifiers, also written M, which create a formula $\mathrm{M}i.\varphi[\psi]$ that is true iff ψ is true for at least half the values of i that make φ true ([MS23a, subsection 2.2]).

5 Complexity Results for Transformers

In this chapter, we will present and prove various complexity results involving transformer networks. For a slightly simplified overview over the majority of the results, see Table 5.1.

	fixed	logarithmic	unbounded
Hard Attention	$\subseteq \mathrm{AC}^0$	$\subseteq \mathrm{AC}^0$	$\subseteq \mathrm{AC}^0$
(Weak/Strong) Saturated Attention	$\subseteq \mathrm{U_L\text{-}TC}^0$	$\mathbb{F}$: $\subseteq \mathrm{U_L\text{-}TC}^0 \subseteq \mathrm{TC}^0$ $\not\subseteq \mathrm{AC}^0$, $\mathbb{Q}$: $= \mathrm{ALL}$	$= \mathrm{ALL}$
Soft Attention	$\subseteq \mathrm{FOC}[+;\mathrm{MOD}] \not\supseteq \mathrm{U_L\text{-}TC}^0$, $\subsetneq \mathrm{U_L\text{-}TC}^0$	$\mathbb{F}$: $\subseteq \mathrm{U_L\text{-}TC}^0 = \mathrm{FO(M)}$	$= \mathrm{ALL}$

Table 5.1: Results shown in this section ($\mathrm{U_L}$ is short for log-uniform here)

5.1 Hard Attention Transformer Networks are in AC^0

This section is based on [HAF22].

We begin by showing that AC^0 is an upper bound on the class of languages that can be recognized by hard attention transformer networks GUHAT, regardless of the internal datatype or precision.

5.1.1 Definition of Generalized Transformer Networks

For our proof of this, we need an even more generalized definition of what a transformer network is than the one in subsection 2.3.2. We change the following points:

- A symbol $\$ \notin \Sigma$ is appended to the input.
- For the representation of activation values, we use an arbitrary set $\mathcal{A}$ rather than $\mathbb{D}^m$.
- The input/embedding function $\varphi\colon (\Sigma \cup \{\$\}) \times \mathbb{N} \times \mathbb{N} \to \mathcal{A}$ takes an extra third parameter $n \in \mathbb{N}$ that will always be the length of the input (including the \$). So, $v_{0,i} = \varphi(w_i, i, n)$.
- The scoring functions $s_{\ell,h}\colon \mathcal{A} \times \mathcal{A} \to \mathbb{R}$ map to real numbers without loss of generality. So, $a_{\ell,h,i,j} = s_{\ell,h}(v_{\ell,i}, v_{\ell,j})$.

- There is a new pooling function $p\colon A^* \times \mathbb{R}^* \to \mathcal{A}$ which combines the attention function α and the weighted sum of the values. For the case of hard attention p^{UHA} is defined as follows: On inputs $(v_0, v_1, \ldots, v_{n-1}) \in \mathcal{A}^n$ and $(a_0, a_1, \ldots, a_{n-1}) \in \mathbb{R}^n$, let $j \in [n]$ be the smallest index that maximizes a_j. Then $p^{\mathrm{UHA}}((v_0, v_1, \ldots, v_{n-1}), (a_0, a_1, \ldots, a_{n-1})) = v_j$. So,

$$b_{\ell,h,i} = p^{\mathrm{UHA}}((v_{\ell,0}, v_{\ell,1}, \ldots, v_{\ell,n-1}), (a_{\ell,h,i,0}, a_{\ell,h,i,1}, \ldots, a_{\ell,h,i,n-1})).$$

- The activation functions $f_\ell\colon \mathcal{A} \times \mathcal{A}^H \to \mathcal{A}$ take a vector of attention values rather than numbers as their second parameter. So,

$$v_{\ell+1,i} = f_\ell(v_{\ell,i}, (b_{\ell,0,i}, b_{\ell,1,i}, \ldots, b_{\ell,H-1,i})).$$

- There is a new model output function $g\colon \mathcal{A} \to [2]$. The output of the generalized transformer network $T(x)$ is computed by applying this function to the last symbol of v_ℓ, $T(x) \coloneqq g(v_{\ell,n-1})$.

5.1.2 A Normal Form for Generalized Hard Attention Transformer Networks

Despite the abstractness and generality of the GUHAT model, we can define a normal form representation and show that every transformer network $T \in \mathrm{GUHAT}$ is equivalent to a transformer network in GUHAT in this normal form with the same number of layers and heads. The key idea is to preserve all the information from previous layers that has been used to compute them in the activation values by requiring that the input and activation functions just return the tuple of their arguments. We also require that attention values be integers in the smallest relevant range.

Definition 5.1. *A GUHAT with L layers and h heads is in* informative normal form *iff the following conditions are satisfied:*

- *The input function is $\varphi(\sigma, i, n) = (\sigma, i, n)$.*
- *For each layer $\ell \in [L]_{+1}$, the activation values are $(H+1)$-tuples of activation values at layer $\ell - 1$, and the activation function is defined by*

$$f_\ell(v, (b_0, b_1, \ldots, b_{H-1})) = (v, (b_0, b_1, \ldots, b_{H-1})).$$

- *For each layer $\ell \in [L]_{+1}$ and attention head $h \in [H]$, the scoring function $s_{\ell,h}$ returns an integer in $[N]$, where N is the total number of possible ordered pairs of activation values at layer $\ell - 1$.*

Lemma 5.2. *For any transformer network $T \in$ GUHAT, there exists a transformer network $\hat{T} \in$ GUHAT in informative normal form such that $L(T) = L(\hat{T})$. Moreover, $\hat{T}$ has the same number of layers and heads as T.*

Proof. Let T be a GUHAT with L layers and H heads, with input alphabet Σ, input function φ, scoring functions $s_{\ell,h}$, activation functions f_ℓ, and output function g. We describe how to construct functions for an equivalent transformer network $\hat{T}$ in GUHAT in informative normal form, which also has L layers and H heads. We assume that n is the input length.

For $\hat{T}$, the input function $\hat{\varphi}(\sigma, i, n)$ is defined to return the triple (σ, i, n). Note that there are at most $|\Sigma| \cdot n$ possible initial activation values. We also define a function t_0 that translates initial activation values for $\hat{T}$ into initial activation values for T by $t_0(\sigma, i, n) = \varphi(\sigma, i, n)$.

Now, we perform induction on the layers of T and $\hat{T}$. Assume that we have defined scoring and activation functions for $\hat{T}$ for layers before ℓ (where the initial activation values are treated as layer 0), and a translation function $t_{\ell-1}$ that translates all possible activation values for $\hat{T}$ from the previous layer into activation values for T from the previous layer. To define the scoring function for $\hat{T}$ for layer ℓ and head h, we enumerate all the possible pairs $\hat{v}_i$ and $\hat{v}_j$ of activation values of $\hat{T}$ at layer $\ell - 1$, and determine the corresponding attention values of T, which we denote by $y_{\ell,h}(\hat{v}_i, \hat{v}_j) = s_{\ell,h}(t_{\ell-1}(\hat{v}_i), t_{\ell-1}(\hat{v}_j))$. We make a list of all the distinct resulting values and sort them in increasing order. Then we define $\hat{s}_{\ell,h}(\hat{v}_i, \hat{v}_j)$ to be the index of $y_{\ell,h}(\hat{v}_i, \hat{v}_j)$ in this sorted list. The activation function for $\hat{T}$ for layer ℓ is, by definition,

$$\hat{f}_\ell(v, (b_0, b_1, \ldots, b_{H-1})) = (v, (b_0, b_1, \ldots, b_{H-1})).$$

The translation function for layer ℓ is defined by

$$\hat{t}_\ell(v, (b_0, b_1, \ldots, b_{H-1})) = f_\ell(t_{\ell-1}(v), (t_{\ell-1}(b_0), t_{\ell-1}(b_1), \ldots, t_{\ell-1}(b_{H-1}))),$$

that is, we translate each of the component activation values using $t_{\ell-1}$ and then apply the activation function of T.

Finally, the output function for $\hat{T}$ is defined by $\hat{g}(\hat{v}) = g(t_L(\hat{v}))$, that is, we translate the layer L activation value $\hat{v}$ of $\hat{T}$ to the layer L activation value of T, and apply the output function of T.

By construction, $\hat{T}$ is in informative normal form, and it has L layers and H heads. It is not difficult to see that for any input w, the translations $t_k(\hat{v})$ of the activation values $\hat{v}$ of $\hat{T}$ are equal to the corresponding activation values of T, and the outputs $\hat{T}(w) = T(w)$ are equal as well. Thus $L(\hat{T}) = L(T)$. □

5.1.3 From GUHAT to Circuits

In this subsection, we show that for every language $L \in \text{GUHAT}$, we can construct a family of Boolean circuits of constant depth and polynomial size that also recognizes L. The key step of the proof is to bound the number of bits needed to represent scoring and activation values for an input sequence of length n by $O(\log(n))$, where the suppressed constants depend on L and H.

Lemma 5.3. *Let T be a GUHAT in informative normal form with L layers and H heads, and alphabet Σ. Let $s = \log(|\Sigma| + 1)$. Then for any input of length n and any $\ell \in [L]$, the activation values at layer ℓ can be represented by $(H+1)^\ell \cdot (2 \cdot \log(n) + s)$ bits, and for $\ell \in [L]_{+1}$, the attention scores at layer ℓ can be represented by $2 \cdot (H + 1)^{\ell-1} \cdot (2 \cdot \log(n) + s)$ bits.*

Proof. For an input sequence of length n, the initial activation values are (σ, i, n), where $\sigma \in \Sigma \cup \{\$\}$ and $i \in [n]$. This can be represented by a string of $2 \cdot \log(n) + s$ bits. At each successive layer, the activation values are a tuple of $(H + 1)$ values from the previous layer, which multiplies the number of bits required to represent them by $(H + 1)$. Also, the range of scoring values is bounded by the number of ordered pairs of activation values at the previous layer, so scoring values can be represented by twice the number of bits to represent an activation value at the previous layer. □

It is worth observing that the bounds provided by Lemma 5.3 do not hold in the case of saturated attention because activation values may be the result of the average of an arbitrary subset of the possible input, which means that there are exponentially more possible activation values at each layer.

The following elementary facts about Boolean circuits will be useful:

Lemma 5.4. *An arbitrary Boolean function $f\colon [2]^n \to [2]^m$ of n inputs and m outputs can be computed by a depth 3 circuit of size at most $2^n + n + m$.*

Proof. We express each output z_i of f as a disjunctive normal form (DNF) formula of at most 2^n terms, each with at most n literals. Then, we construct the circuit from an AND gate for each of the 2^n possible DNF terms, requiring a NOT gate for each of the n input bits, and an OR gate for each of the m output bits taking the AND gates corresponding to the terms of the DNF as its inputs. The resulting circuit has size $2^n + n + m$. The longest possible path to an output from an input is through a NOT, an AND and the OR gate, for a depth of at most 3. □

Corollary 5.5. *If a Boolean function f has at most $c \cdot \log(n)$ inputs and at most $d \cdot \log(n)$ outputs, then it may be computed by a Boolean circuit of depth 3 and size at most $n^c + c \cdot \log(n) + d \cdot \log(n)$.*

We now have the necessary tools to prove the following theorem:

Theorem 5.6. *Every language in* GUHAT *is recognizable by a family of circuits in* AC^0.

Proof. Let L be a language over Σ that is in GUHAT. By Lemma 5.2, we may assume that L is recognized by a GUHAT transformer network T in informative normal form. Assume T has L layers and H heads.

What we describe below is a family of circuits to recognize the end-marked language $L\$$, which can easily be converted to a family of circuits that recognizes L by hard-wiring the representation of the end-of-sequence symbol \$ at the end of the input string using constant gates. Let $s = \log(|\Sigma| + 1)$ and let h be any binary symbol encoding for $\Sigma \cup \{\$\}$. We construct a family of Boolean circuits $(C_{s,n})_{n\in\mathbb{N}}$ of constant depth and polynomial size such that for all positive integers n and all $w \in \Sigma^{n-1}$, $w \in L$ iff $C_{s,n}(h(x\$)) = 1$.

With the $O(\log(n))$ bound on the number of bits to represent activation and scoring values, Lemma 5.3 yields circuits of constant depth and size polynomial in n for the input, scoring, activation, and output functions. Additional circuitry is necessary to implement the comparison of attention scores and selection of the activation value to attend to for each position, layer, and head.

We construct the overall circuit $C_{s,n}$ according to the layers of T, starting with the input function. Let the inputs to T be w_i for $i \in [n]$. The inputs to $C_{s,n}$ are $w_{i,j}$ for $i \in [n]$ and $j \in [s]$, where $w_{i,j}$ are the bits of $h(w_i)$, representing the binary encoding of input symbol w_i. At layer 0 for position i, the value of $v_i^{(0)} = \varphi(w_i, i, n) = (w_i, i, n)$ is achieved using the input wires $w_{i,j}$ for $j \in [s]$ followed by a sequence of constants 0 or 1 representing the binary representations of i and n for a total of $2 \cdot \log(n) + s$ wires representing the value (w_i, i, n).

Performing induction on layers, we assume that for some $\ell \in [L]_{+1}$ the circuit $C_{s,n}$ has been constructed to contain the wires representing all the activation values $v_i^{(\ell-1)}$ for $i \in [n]$ at layer $\ell - 1$. The portion of the circuit computing the representation of activation values at layer ℓ is described as follows: Fix a position $i \in [n]$ and a head $h \in [H]$. For each $j \in [n]$, there is a circuit $A_{\ell,h,i,j}$ that has as input the wires for the activation values $v_i^{(\ell-1)}$ and $v_j^{(\ell-1)}$ and as output wires representing the natural number attention score $a_{\ell,h,i,j}$ in binary. Each of these circuits $A_{\ell,h,i,j}$ has $2 \cdot (H+1)^{\ell-1} \cdot (2 \cdot \log(n) + s)$ inputs and outputs by Lemma 5.3, and therefore can be computed using depth 3 and size polynomial in n, by Corollary 5.5. All $H \cdot n^2$ such circuits for layer ℓ operate in parallel, for overall depth 3 and size polynomial in n.

We next describe the circuit that implements the pooling function f^{UHA}. For each pair $j, j' \in [n]$, there is a circuit $D_{\ell,h,i,j,j'}$ whose inputs are the outputs of $A_{\ell,h,i,j}$ and $A_{\ell,h,i,j'}$ and whose output is a single wire $g_{\ell,h,i,j,j'}$ with a value of 1 if $a_{\ell,h,i,j} \geq a_{\ell,h,i,j'}$ and

0 otherwise. Because of the bounds on the number of inputs and outputs, each of these circuits can have depth 3 and size polynomial in n by Corollary 5.5. These n^2 circuits all compute in parallel. Then for each position j, whether j maximizes $a_{\ell,h,i,j}$ can be computed by an AND gate whose inputs are $g_{\ell,h,i,j,j'}$ for all $j' \in [n]$. Let the output of this AND gate be denoted $m_{\ell,h,i,j}$. Then $m_{\ell,h,i,j} = 1$ iff the position j maximizes $a_{\ell,h,i,j}$. This increases the depth by 1.

For each j, an indicator $z_{\ell,h,i,j}$ is computed by an AND gate whose inputs are $m_{\ell,h,i,j}$ and $\neg(m_{\ell,h,i,j'})$ for all $j' < j$. Thus, $z_{\ell,h,i,j} = 1$ iff j is the leftmost position that maximizes $a_{\ell,h,i,j}$. This increases the depth by 2.

Finally, these indicator values are used to combine the layer $\ell - 1$ activation values in a selection circuit, yielding the representation of the activation value $b_{\ell,h,i} = v_j^{(\ell-1)}$ such that $z_{\ell,h,i,j} = 1$. In general, such a selection circuit takes as input t selector bits $z_0, z_1, \ldots, z_{t-1}$, where exactly one $z_j = 1$, and t input values $w_0, w_1, \ldots, w_{t-1}$, where each w_r consists of S bits. It outputs S bits representing the selected w_j (for which $z_j = 1$). Letting $w_{r,s}$ denote the bit s of w_r, the computation can be described as $v_{r,s} = w_{r_s} \wedge z_r$ for $r \in [t]$ and $s \in [S]$, which can be computed by one layer of $t \cdot S$ AND gates in parallel. Then the bits of the output are $u_s = \bigvee_{r \in [S]} v_{r,s}$ for $s \in [S]$, which can be computed by one layer of S OR gates in parallel. Thus, the selection circuit adds 2 to the depth, and a polynomial in n to the size.

Because each activation function for a GUHAT in informative normal form simply returns its argument, no further computation is needed for the activation values. The representation of the activation value $v_i^{(\ell)}$ is just the sequence of wires representing $v_i^{(\ell-1)}$, followed by those representing $b_{\ell,0,i}$ through $b_{\ell,H-1,i}$.

To produce the output of the circuit, we note that the representation of $v_n^{(L)}$ has $O(\log(n))$ bits and the output of g is a single bit, so g can be implemented by a Boolean circuit of constant depth and size polynomial in n, by Corollary 5.5. This concludes the proof. □

5.2 Universality of Saturated Transformer Networks

This section is based on [MSS22].

In order to formulate upper bounds to the power of saturated attention transformers, we need to constrain their internal functions. Our definition of the class of languages recognizable by saturated transformers AHAT($\mathbb{D}$) already includes the restriction that they are size-preserving. This, however, is not yet sufficient to allow for a nontrivial upper bound, as we will show that saturated transformers are still able to recognize any formal language. Our proof of this works by encoding the entire input sequence into a single value and using the activation block as a black box to recognize the language.

Theorem 5.7. $\mathrm{AHAT}(\mathbb{Q}) = \mathrm{ALL} = \mathcal{P}([2]^*)$.

Proof. Let $L \in \mathrm{ALL}$ be any formal language over the alphabet $\Sigma = [2]$. We construct a rational-valued saturated transformer network with 1 layer and 1 head to recognize L. We will omit ℓ and h subscripts. Let p_i denote the i-th prime number. The embedding layer encodes the position i of each token $w_i \in \Sigma$ according to

$$\varphi(w_i, i) \coloneqq \frac{w_i}{p_i}.$$

Since $p_i \sim i \cdot \log(i)$ for large i by the prime number theorem [Gol73, p. 599], the number of bits needed to represent the denominator of $\varphi(w_i, i)$ is bounded by

$$p_i \leq c \cdot \log(i \cdot \log(i)) \leq c \cdot \log(i^2) = 2c \cdot \log(i)$$

for some constant factor c. As i has size $\log(i)$, this implies that φ is size-preserving. Now we define a single uniform attention head that sums all v_i, outputting

$$\sum_i v_i = \sum_i \varphi(w_i, i) = \sum_{i, w_i = 1} \frac{1}{p_i}.$$

The denominator q of this sum is the product $\prod_{i, w_i = 1} p_i$, which can be shown by induction over the number of prime numbers that are multiplied. Note that $w_i = 1$ iff p_i divides q. Thus, we can define a function g that extracts the input sequence w from q by checking for each i whether p_i divides q. We let

$$c_L(w) = \begin{cases} 1, & w \in L \\ 0, & \text{otherwise} \end{cases}$$

be the characteristic function of L and set $f \coloneqq c_L \circ g$. The output of the transformer network will now compute whether $w \in L$, since g outputs the original input sequence w and c_L decides whether $w \in L$. Note that any function solving a decision problem has a codomain of fixed size, namely the set $[2]$, and is therefore size-preserving. □

Similar to this, other authors have used arbitrary precision for storing and manipulating positional encodings to show their model of transformer networks to be Turing complete [PMB19, PBM21].

Both the unnatural construction that encodes positions using prime numbers and the result that they can decide any formal language strongly indicate that these restrictions on our model of transformer networks are not yet sufficient to mimic reality. Because of this, we switch the datatype from rationals to floats. In the following section, we will see that doing this allows us to bound the capabilities of saturated transformer

networks in TC^0.

But before we do that, we will quickly show that using floats alone is not enough and size-preservation is also needed to be able to find non-trivial upper bounds. The unbounded prefix added to AHAT($\mathbb{D}$) simply means that we no longer require the internal functions to be size-preserving.

Corollary 5.8. unbounded-AHAT($\mathbb{D}$) = ALL *for* $\mathbb{D} \in \{\mathbb{F}, \mathbb{Q}\}$.

Proof. The proof works exactly the same as the proof for Theorem 5.7, with the exception that if $\mathbb{D} = \mathbb{F}$ the prime numbers p_i need to be replaced with distinct powers of 2. This needs to be done because floats can only have powers of 2 in the denominator. The removal of the size bound allows us to use the powers of 2 that grow in size linearly instead of logarithmically. We can once again define a function g that extracts the input sequence by just looking at which bits are set to 1. This completes the proof. □

Corollary 5.9. unbounded-SAT($\mathbb{D}$) = ALL *for* $\mathbb{D} \in \{\mathbb{F}, \mathbb{Q}\}$.

Proof. The proof works exactly the same as the proof for Corollary 5.8. Soft attention transformers with sufficient (or, in this case, unbounded) precision can implement uniform attention by setting all queries and keys to be constant [MS23a, p. 5]. □

5.3 Saturated Attention Transformer Networks are not in AC^0

This section is based on [MSS22].

We define the MAJ $\coloneqq \{w \in [2]^* \mid |w|_1 > |w|_0\}$ decision problem similarly to the MAJ^n functions in Definition 3.3. It was first shown that transformer networks can recognize MAJ by [PMB19, Proposition 3.3]. We will prove that this still holds true for saturated transformer networks using floats by using only a single uniform attention head. As MAJ $\notin \mathrm{AC}^0$ by Smolensky's Theorem [Vol99, Theorem 3.31], this shows that saturated transformer networks are not in AC^0.

Theorem 5.10. AHAT($\mathbb{F}$) $\not\subseteq \mathrm{AC}^0$.

Proof. We will construct a single layer transformer network with a single attention head to recognize MAJ, omitting the ℓ and h subscripts. Let the embedding function

$$\varphi(w_i, i) \coloneqq (1 - w_i, w_i) = \begin{cases} (1,0), & w_i = 0 \\ (0,1), & w_i = 1 \end{cases}$$

be a 1-hot encoding of w_i. Set the scoring function $s(x_i, x_j) \coloneqq 1$ for all inputs $x_i, x_j \in \mathbb{F}^2$, resulting in the attention head attending everywhere and computing for every

$i \in [n]$: $b_i = \left(\frac{|w|_0}{n}, \frac{|w|_1}{n}\right)$. Finally, set

$$f(v_i, b_i) \coloneqq \begin{cases} 1, & b_{i,1} > b_{i,0} \\ 0, & \text{otherwise} \end{cases}, \text{ where } b_i = (b_{i,0}, b_{i,1}).$$

Thus, the output of the transformer network is all 1s if $|w|_1 > |w|_0$, or $w \in \text{MAJ}$, and all 0s otherwise. We have shown $\text{MAJ} \in \text{AHAT}(\mathbb{F})$, and, with $\text{MAJ} \notin \text{AC}^0$, it directly follows that $\text{AHAT}(\mathbb{F}) \not\subseteq \text{AC}^0$. □

Note that this construction is not just possible in our generalized transformer network model, but can also be implemented in transformer networks that are actually in use, like the original one described in section 2.2. It has also been shown empirically that single layer transformer networks can learn to recognize the majority language.

5.4 Saturated Attention Transformer Networks are in TC^0

This section is based on [MSS22].

Lemma 5.11. *Let $v_0, v_1, \ldots, v_{n-1}$ be a sequence of floats, each with size at most z. Then their sum $s = \sum_{i \in [n]} v_i$ has size at most $4z + 2\log(n) + 1$.*

Proof. Let p_i, q_i and p_s, q_s denote the numerator and denominator of v_i and s, respectively. Since each v_i has size at most z, both p_i and q_i also have size at most z. Let $p_{\max} = \max_i p_i$ and $q_{\max} = \max_i q_i$. To add these floats, all their denominators have to be made equal to $q_{\max}$, which results in their numerators also being multiplied by $\frac{q_{\max}}{q_i}$. This works because q_i divides $q_{\max}$, since they are both powers of 2. We can now estimate the numerator of s as

$$p_s \leq \sum_{i \in [n]} p_i \cdot \frac{q_{\max}}{q_i} \leq n \cdot p_{\max} \cdot q_{\max}$$

which has size $\leq \log(n) + z + z = 2z + \log(n)$. The denominator $q_s \leq q_{\max}$ has size $\leq z$. Therefore, s has size $\leq 1 + 2 \cdot \max(2z + \log(n), z) = 4z + 2\log(n) + 1$. □

In particular, the size of the sum of a sequence of n float values whose size is bounded by $z(n) \in O(\log(n))$ is also bounded by $O(\log(n))$.

We will now leverage Lemma 5.11 to show that over any transformer network over floats with an element-wise-size-preserving attention function, the values are of bounded size.

Definition 5.12. *A function* $\alpha\colon \mathbb{D}^n \to \mathbb{D}^n$ *is* element-wise-size-preserving *iff for* $i \in [n]$ *the function* $x_i \mapsto \alpha(x)_i$ *is size-preserving, where* $x \in \mathbb{D}^n$.

Note that saturated attention satisfies this definition. We can now prove a theorem bounding the size of the representations in transformer networks with element-wise-size-preserving attention.

Theorem 5.13. *For any transformer network over* $\mathbb{F}$ *with* $\varphi, s_{\ell,h}, f_\ell \in \mathcal{P}$ *and* α *element-wise-size-preserving, for all* $\ell \in [L+1]$, $h \in [H]$, *and* $i \in [n]$, $v_{\ell,i}$ *has size* $O(\log(n))$.

Proof. By induction over ℓ. The proof follows the definition of transformer network computation in subsection 2.3.2.

Base case ($\ell = 0$): w_i has size $O(1)$ and $i \in [n]$ has size $O(\log(n))$. Since $\varphi \in \mathcal{P}$, $v_{0,i} = \varphi(w_i, i)$ has size $O(\log(n))$ for all $i \in [n]$.

Inductive Step: Assuming $v_{\ell,i}$ has size $O(\log(n))$, we will show that $v_{\ell+1,i}$ does too. As $s_{\ell,h} \in \mathcal{P}$, $a_{\ell,h,i,j} = s_{\ell,h}(v_{\ell,i}, v_{\ell,j})$ has size $O(\log(n))$ for all $i, j \in [n]$. Since α is element-wise-size-preserving, we can conclude that $\alpha(a_{\ell,h,i,0}, \ldots, a_{\ell,h,i,n-1})_j$ also has size $O(\log(n))$ for all $h \in [H], i, j \in [n]$. Multiplying two floats is also size-preserving [MSS22, Appendix B], so $\alpha(a_{\ell,h,i,0}, \ldots, a_{\ell,h,i,n-1})_j \cdot v_{\ell,j}$ has size $O(\log(n))$ for all $h \in [H]$ and $i, j \in [n]$. We then apply Lemma 5.11 to conclude that $b_{\ell,h,i}$ has size $O(\log(n))$, where, recall,

$$b_{\ell,h,i} = \sum_{j \in [n]} \alpha(a_{\ell,h,i,0}, \ldots, a_{\ell,h,i,n-1})_j \cdot v_{\ell,j}.$$

Finally, computing $v_{\ell+1,i} = f_\ell(v_{\ell,i}, (b_{\ell,0,i}, \ldots, b_{\ell,h-1,i}))$, we conclude that $v_{\ell+1,i}$ has size $O(\log(n))$ for all i due to the size-preservation by f_ℓ. □

Corollary 5.14. *For any saturated transformer network over* $\mathbb{F}$ *with size-preserving internal functions, for all* $l \in [L+1]$ *and* $i \in [n]$, $v_{\ell,i}$ *has size* $O(\log(n))$.

Proof. This follows from Theorem 5.13 because saturated attention is element-wise-size-preserving. □

This result cannot be applied to soft attention because the softmax function is not guaranteed to be element-wise-size-preserving as it involves computing the exponential function.

We have proved that each vector in a saturated transformer network over floats has size $O(\log(n))$. Now, we show how this implies saturated transformer networks can be simulated by TC^0 circuits.

Corollary 5.15. *Any size-preserving function with at most* $c \cdot \log(n)$ *input bits can be computed by a Boolean circuit of depth* 3 *and polynomial size.*

Proof. This follows directly from Lemma 5.4. □

In other words, such functions can be computed by AC^0 circuits. In addition, we will show that the sum of n floats of size at most $c \cdot \log(n)$ can be computed by TC^0 circuits.

Lemma 5.16. *Let $v_0, \ldots, v_{n-1}$ be a sequence of floats with size at most $c \cdot \log(n)$ for some c. Then their sum $s = \sum_{i \in [n]} v_i$ is computable by a threshold circuit of constant depth and polynomial size.*

Proof. Let p_i, q_i once again be the numerator and denominator of v_i. We first compute $q_{\max} = \max_i q_i$ using an AC^0 circuit that compares all pairs q_i, q_j and returns the first q_i such that $q_i \geq g_j$ for all $j \in [n]$. We then use the fact that multiplication and right shift (q_i is a power of 2) are in TC^0, in order to compute $r_i \coloneqq p_i \cdot \frac{q_{\max}}{q_i}$ in parallel for all $i \in [n]$. Note that q_i and $q_{\max}$ are both powers of 2, so division will be exact. Next, we leverage the fact that the sum of n integers with size $O(\log(n))$ is in TC^0, in order to compute the numerator of the sum $p' = \sum_i r_i$. We select the denominator as $q' = q_{\max}$. Finally, we add an AC^0 circuit that reduces the fraction by removing shared trailing 0s from p' and q', which is possible by Corollary 5.15. Thus, we have constructed a TC^0 circuit to compute the sum of n floats of size $O(\log(n))$. □

We now construct a TC^0 circuit that simulates a saturated transformer network over floats.

Theorem 5.17. $\text{AHAT}(\mathbb{F}) \subseteq \text{TC}^0$.

Proof. For each n, we construct a TC^0 circuit that simulates a saturated transformer network of input size n. We construct the circuit modularly, with one subcircuit for the attention mechanism and another for the feedforward subnetwork.

Attention Head: Fix a single head in some layer. We will construct a TC^0 subcircuit that simulates the attention mechanism at position i. The head attends over vectors $v_0, \ldots, v_{n-1}$. For all $j \in [n]$, v_j has size $O(\log(n))$ by Theorem 5.13. In parallel for each j, we compute the scores $a_{i,j} = s(v_i, v_j)$ with an AC^0 circuit by Corollary 5.15. We then compute $a_{i,\max} \coloneqq \max_j a_{i,j}$ with an AC^0 circuit by comparing all v_j pairwise and selecting the first v_k such that $v_k \geq v_j$ for all $j > n$. We then compute "masked" values $u_{i,j}$ for each $j \in [n]$ via an AC^0 circuit by Lemma 5.4:

$$u_{i,j} \coloneqq \begin{cases} v_j, & a_{i,j} \geq a_{i,\max} \\ 0, & \text{otherwise.} \end{cases}$$

We then compute the sum $s_i \coloneqq \sum_{j \in [n]} u_{i,j}$ by Lemma 5.16. By Lemma 5.11, s_i has size

$O(\log(n))$. Now, we similarly define

$$z_{i,j} := \begin{cases} 1, & a_{i,j} \geq a_{i,\max} \\ 0, & \text{otherwise.} \end{cases}$$

Using an analogous sum construction with $z_{i,j}$ instead of $u_{i,j}$, we can use a TC^0 circuit to compute $|\mathcal{M}(a)|$: the number of values of j for which $a_{i,j} \geq a_{i,\max}$. Finally, since dividing floats is in TC^0 [MSS22, Appendix A], we can compute the head output as $\frac{s_i}{|\mathcal{M}(a)|}$, which has size $O(\log(n))$ by size preservation of division.

Feedforward: As input, f receives v_i as well as H head outputs, all of which have size $O(\log(n))$. As the total size of the input is $O(\log(n))$, we can use Corollary 5.15 to compute the output of f with an AC^0 circuit. The size of the output is $O(\log(n))$ by size preservation of f. The same idea holds for φ as well as the linear classification head.

We have simulated each transformer network component with a TC^0 subcircuit, completing the proof. □

A uniform version of this result has been shown in [Str23]. We will not go into further detail on this right here, since we will show a more general uniform result in section 5.6.

5.5 Fixed Precision Transformer Networks are in FOC[+; MOD]

This section is based on [CCP23].

In this subsection, we will show an upper bound on the expressivity of fixed-precision transformer networks. A fixed-precision transformer network is one where the internal functions are not size-preserving, but instead the size of their output is bounded by a constant called the precision.

We are going to use fixed-point numbers instead of floats. There is no loss of generality because all floats can be converted exactly to a fixed point number. This fixed point number may be significantly larger, but will still be bounded by some constant precision.

A transformer network computes many activations, or internal values, that depend on the input w and can be thought of as functions $a\colon \Sigma^* \to \mathbb{FP}$. For each such activation, we will write sentences in FOC[+; MOD] that test the bits of $a(w)$.

Definition 5.18. *If $a\colon \Sigma^* \to \mathbb{FP}_{r,s}$, we say that a is* defined *by sentences $(\sigma_k^a)_{k\in[r+s]}$ (or (σ_k^a) for short) iff, for all $k \in [r+s]$, $w \models \sigma_k^a \Leftrightarrow a(w)_k = 1$.*

Similarly, if $a\colon \Sigma^n \to \mathbb{FP}^n$, we say that a is defined *by $(\varphi_k^a[p])$ iff, for each $p \in [n]$, $[a(w)]_p$ is defined by $(\varphi_k^a[p])$.*

The finiteness of $\mathbb{FP}$ ensures the following fact:

Lemma 5.19. *If $a\colon \Sigma^* \to \mathbb{FP}$ is defined by (σ_k^a), then for any function $f\colon \mathbb{FP} \to \mathbb{FP}$ there are sentences that define $f \circ a$. Similarly, if $b\colon \Sigma^* \to \mathbb{FP}$ is defined by (σ_k^b) and $g\colon \mathbb{FP} \times \mathbb{FP} \to \mathbb{FP}$, there are sentences that define the function $g \circ (a, b) = \{w \mapsto g(a(w), b(w))\}$.*

Proof. Because $\mathbb{FP}$ is finite, it is easy but tedious to write sentences that test for all possible inputs and outputs. □

Using this, the following result can be shown:

Theorem 5.20. *Every language that is recognizable by a fixed-precision transformer network is definable by a sentence of* $\mathrm{FOC}[+;\mathrm{MOD}]$.

Proof Sketch. The proof works by going through the components of a transformer network and defining their output as an activation function using the output of the previous component(s). Lemma 5.19 allows for this to work as long as no over- or underflows occur during the computation. Because of this, the internal functions are not allowed to be arbitrary, but have to follow the definition of the original transformer network more closely. For the same reason, the softmax function has to be more carefully computed by bitwise averages instead of sums. With these changes, it is then possible to create a $\mathrm{FOC}[+;\mathrm{MOD}]$ sentence that defines the output activation function of the transformer network. For the full details of the proof, see [CCP23, section 5]. □

This result is particularly interesting as $\mathrm{FOC}[+;\mathrm{MOD}]$ is not a superset of log-uniform TC^0 and thus yields a tighter upper bound.

Theorem 5.21. *The language $\{0^n 1^n \mid n \in \mathbb{N}\}$ is in* log*-uniform TC^0, but not definable in* $\mathrm{FOC}[+;\mathrm{MOD}]$.

Proof. The language contains no words of odd length and exactly one word of each even length. Because of this, a circuit family that decides it can be constructed from circuits that simply output a constant 0 for an odd amount of input bits and circuits that test for $0^n 1^n$ for an even amount of input bits $2 \cdot n$. Such a circuit just consists of one AND gate that takes the first n input bits negated and the other input bits directly as its inputs. An example of this can be seen in Figure 5.1. This circuit family clearly lies in log-uniform TC^0.

To show that the language is not definable in $\mathrm{FOC}[+;\mathrm{MOD}]$, suppose that it is definable by some sentence σ. Let M be the product of all moduli m used in atomic formulas $\mathrm{MOD}_r^m(p)$ used in σ. Then σ cannot distinguish between positions p and $p + M$, so it cannot distinguish $w = 0^M 1^M$ and $w' = 10^{M-1}01^{M-1}$. Since $w \models \sigma$, it must be the case that $w' \models \sigma$, which is a contradiction. □

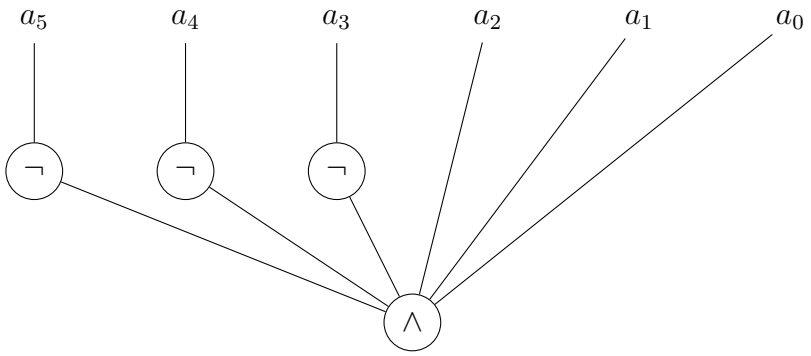

Figure 5.1: the circuit that checks if a given word $a \in [2]^6$ is 000111

The authors of [CCP23] continue by showing that every language that is definable by a sentence in FOC[+; MOD] is also recognizable by a transformer network. They argue that FOC[+; MOD] lies somewhere between fixed-precision transformer networks and unbounded transformer networks, and is therefore close to an exact characterization of the languages that transformer networks can recognize.

This result, however, is really quite weak, as we already showed in Corollary 5.9 that unbounded-SAT($\mathbb{D}$) = ALL, which is not close to FOC[+; MOD] at all. Their definitions and assumptions do not line up exactly with the ones we use here, so our result does not immediately imply theirs, but it makes their reasoning appear somewhat questionable.

5.6 log-Precision Transformer Networks are in Uniform TC^0

This section is based on [MS23b].

In this subsection, we are going to show that log-precision transformer networks can be simulated by uniform constant-depth threshold circuits. Thus, such transformer networks can only solve problems in uniform TC^0.

Intuitively, the upper bound states that log-precision transformer networks are computationally shallow, and that this shallowness can be understood to emerge from their parallelizability. Their inherent parallelism is useful for training them efficiently at massive scale, but may limit the complexity of the computations they can express. The term *parallelism tradeoff* has been introduced in [MS23b, section 1] to capture this idea, which represents a potential fundamental weakness of the current paradigm of scaling language models. One interpretation of complexity classes such as AC^0 and TC^0 is as sets of in polynomial time solvable problems that are parallelizable to a very high degree — they can be solved in parallel in constant time with enough parallel processors. This gives some intuitive explanation of our result: log-precision transformers end up

in TC^0 because they were designed to be highly parallelizable. Since parallelism is an important property of today's dominant paradigm of training models at massive scale, this points to the conclusion that any massively scaled up model — transformer network or otherwise — will likely obey restrictions similar to the ones derived here for log-precision transformer networks. There is thus an important tradeoff between the massive parallelizability of today's networks and their representation power.

Other results rely on making unrealistically strong assumptions or placing unrealistic restrictions on the model of transformer networks. For this result, we only make one assumption – namely, all intermediate values in the transformer network are limited to $O(\log(n))$ bits, where n is the number of input tokens. We next discuss some implications of this assumption and what the findings mean for practical transformer networks.

The bounds we will prove are asymptotic in nature and thus apply when n is sufficiently large. In practice, transformer network models use fixed precision at each computation node, which is more restrictive than log-precision. However, this constant could be large and thus, for relatively small n, our results do not rule out practical transformer networks solving difficult problems. The results, however, do show that as n grows sufficiently large, log-precision transformer networks are fundamentally limited to problems within TC^0 and cannot accurately solve various commonly studied problems, such as:

- Linear equalities: find x such that $Ax = b$ (assuming log-uniform $\mathrm{TC}^0 \neq \mathrm{P}$)
- Universal context-free recognition (assuming log-uniform $\mathrm{TC}^0 \neq \mathrm{P}$)
- Propositional satisfiability, SAT (assuming log-uniform $\mathrm{TC}^0 \neq \mathrm{NP}$)
- Horn-clause satisfiability, HORN-SAT (assuming log-uniform $\mathrm{TC}^0 \neq \mathrm{P}$)
- AI planning
- Permanent computation.

Extending our analysis to small n will help close the gap to practice.

The formal model we will use in this section is based on a binary classification view of transformer networks. However, our results apply directly to multi-class classification as well and can be extended to generation problems by viewing, for instance, next word prediction in natural language processing (NLP) as a multi-class classification problem. However, if the transformer network decoder is allowed to condition on its previous output in a generation problem, then his would violate our formal setup.

5.6.1 Circuit Serialization

First, we will discuss a way of serializing a circuit into a string. We later show how to generate such serializations using a resource-bounded algorithm, which is the key to proving containment in uniform complexity classes.

We identify a circuit with its serialization in a formal language that identifies each node's label and adjacency list. We will adopt a specific grammar for concreteness, but our construction can be adapted to other string representations of circuits.

We define a circuit serialization as a traversal of a circuit ordered by some topological sort. In this serialization, leaf nodes (variables/input gates) are represented by the string X. An internal node (non-input gate) is represented in Polish notation by the function it computes (AND, OR, or NOT) followed by a list of pointers to its arguments. Each argument $\&1^j$ of gate i encodes (in unary) a zero-indexed pointer of the j-th gate in the circuit, where $j < i$. The final node is interpreted as the circuit output.

To serialize $\{\wedge, \vee\}$-circuits, we use the following grammar, where the i parameter is passed through Gate[i] non-terminals to track the index of the gate in left-to-right order:

$$\begin{aligned} \text{Circuit} &\to \text{Gate}[1]\ \text{Gate}[2]\ \ldots\ \text{Gate}[g] \\ \text{Gate}[i] &\to \mathsf{X} \mid \mathsf{NOT}\ \text{Arg}[i] \mid \text{Op}\ \text{Arg}[i]^* \\ \text{Arg}[i] &\to \&1^j \text{ such that } j < i \\ \text{Op} &\to \mathsf{AND} \mid \mathsf{OR} \end{aligned}$$

In the Arg[i] rule, we enforce that $j < i$ so that arguments must be pointers to already defined gates. As an example of this serialization language, the circuit for $x_0 \vee \neg x_1 \vee x_2$ which can be seen in Figure 5.2 is represented as X X X NOT &1 OR & &111 &11, where spaces are added for readability.

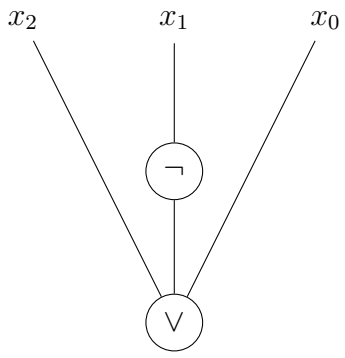

Figure 5.2: the circuit for $x_0 \vee \neg x_1 \vee x_2$

By convention, negations in AC^0 circuits are usually taken to occur at the beginning of the circuit, rather than after $\wedge$ or $\vee$ nodes, which can be achieved using De Morgan's law. Our serialization grammar does not enforce this property, but of course any circuit with this property can be serialized by our grammar.

It is slightly more complicated to serialize threshold circuits. We assume that all non-input gates in our threshold circuits are threshold gates $\theta_{\leq k}$, $\theta_{\geq k}$, which return whether at most or at least k of their m input bits are 1. Threshold gates are equivalent to majority gates (under constant-depth reduction) and can be used to simulate $\wedge$, $\vee$, and $\neg$ gates. Formally, a threshold circuit serialization is generated by the following grammar:

$$\begin{aligned}
\text{Circuit} &\to \text{Gate}[1]\ \text{Gate}[2]\ \ldots\ \text{Gate}[g] \\
\text{Gate}[i] &\to \mathsf{X} \mid \text{Dir}\ 1^k 0^{m-k}\ \text{Arg}[i]^m \\
\text{Arg}[i] &\to \&1^j \text{ such that } j < i \\
\text{Op} &\to <= \mid >=
\end{aligned}$$

In the rule for Gate$[i]$, $m \in \mathbb{N}$ is the arity of the gate, and $k \leq m$ is its threshold. The span 1^k after Dir can be interpreted semantically as a unary encoding of the parameter k for a threshold gate, padded by 0s to the number of total arguments of gate i. For simplicity, we imagine $\neg$ gates are represented as unary $\theta_{\leq 0}$ gates. Thus, the circuit for $\theta_{\geq 1}(x_0, \neg x_1)$ which can be seen in Figure 5.3 would be represented as

$$\mathsf{X}\,\mathsf{X} <= 0\ \&1 >= 10\ \&\ \&11.$$

We say a threshold circuit is in *prefix form* iff all inputs (X) come before all threshold gates ($<=$ and $>=$), as is the case in this example.

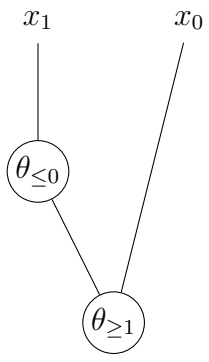

Figure 5.3: the circuit for $\theta_{\geq 1}(x_0, \neg x_1)$

5.6.2 Uniformity

The circuit families we have defined thus far are *nonuniform*, meaning that we do not enforce that the circuits must be related in any way. In degenerate cases, nonuniform circuit families can solve undecidable problems because they have infinite description length, making them a physically unrealizable model of computation. Complexity theorists have thus introduced *uniform* circuit families. Uniform circuit families are a realizable model of computation with relations to classes in computational complexity and formal language theory.

Intuitively, in a uniform circuit family, the circuits for different input sizes must be "somewhat similar" to each other. We formalize this by saying that there exists a resource-constrained Turing machine that maps the input 1^n to a serialization of circuit C_n.

Definition 5.22. *A language L is $(S(n), I(n))$-space uniformly computable by a circuit model M iff there exists a Turing machine that, for all $n \geq 0$, uses $S(n)$ space to map 1^n to an M-circuit recognizing L on inputs of size $I(n)$.*

This notion of uniformity is more general than the standard notion in that the input size $I(n)$ is a function of the problem complexity n. The reason for this is that we will apply uniformity to sub-computations with different input sizes $I(n)$ within a larger computation of input size n. The standard notion of uniformity corresponds to $I(n) = n$.

Furthermore, we will refer to a circuit family as *uniform* iff it is uniformly computable with $S(n) = O(\log(n))$. We can define uniform versions of AC^0 and TC^0 by adopting the previous definitions exactly, but also enforcing uniformity.

5.6.3 Transformer Network Precision and Space

We will assume that each transformer network is resource bounded in terms of the *precision* of each value it computes and, for some of our results, the *space* it uses for the computation of the key operations such as embedding, attention, and activation. Specifically, we will assume precision p, that is., the values at all layers, as well as the outputs of all key intermediate operations in it (attention, activation, arithmetic operators, etc.), are represented using p bits. This is a realistic assumption as, in practice, today's transformer networks are typically limited to the 64-bit precision of the underlying hardware. Formally, we define p-precision as follows:

Definition 5.23. *A k-ary function $f\colon x_0, x_1, \ldots, x_{k-1} \mapsto y$ is p-*precision *iff $x_0, x_1, \ldots, x_{k-1}, y \in [2]^*$ have size at most p bits, and f can be computed by a p-space-bounded Turing machine.*

This means the size of the function inputs and output are bounded by p. Similarly, the intermediate space used by the computation must also be bounded by p. Thus, higher precision computations cannot somehow be hidden inside f.

Definition 5.23 naturally applies to functions with bounded arity k. We will also need to define p-precision for the summation operator in the transformer network, which adds n different floats of size p. Adding n floats can blow up the precision needed to represent their sum. For example, imagine adding the floats $1 \cdot 2^0 + 1 \cdot 2^c$. We obtain $(2^c + 1) \cdot 2^0$, whose mantissa takes $c + 1$ bits to represent. In practice, computers do not preserve full precision in such situations. Instead, small terms like $1 \cdot 2^0$ are discarded. Thus, we define the transformer network's addition operator $\oplus$ to be similarly approximate. For more details on how (iterated) addition of p-precision floats works, see [MS23b, Appendix A].

5.6.4 p-Precision Transformer Network Definition

For this section, we will use the following model of transformer networks. We define an attention head as follows:

Definition 5.24. *A p-precision attention head is specified by a binary p-precision* similarity *(or attention) function $s\colon [2]^p \times [2]^p \to [2]^p$.*

Let $h_0, h_1, \ldots, h_{n-1} \in [2]^p$ be the input sequence to a p-precision attention head, and let $\oplus$ be approximate floating-point addition.

Definition 5.25. *For all $\ell \geq 0$, a p-precision attention head $H_h^{\ell+1}$ computes a vector $a_{i,h}^{\ell+1} \in [2]^p$ via*

$$a_{i,h}^{\ell+1} = \bigoplus_{j \in [n]} \frac{s\big(h_i^\ell, h_j^\ell\big)}{Z_i} \cdot h_j^\ell,$$

where $Z_i = \bigoplus_{j \in [n]} s\big(h_i^\ell, h_j^\ell\big)$.

Standard attention heads, like the ones of the original transformer network, are a special case of this definition where s is scaled dot-product similarity between keys and queries. Standard transformer networks also have a linear or affine value function applied to each head h_j^ℓ in the sum over the j. By its affineness, the value function can, without loss of generality, be removed from the attention head and considered to be a part of the transformer network layer (that is, applied to the output of the attention head).

A p-precision transformer network layer is then a tuple of heads and a function f used to combine them.

Definition 5.26. *A p-precision transformer network layer is a tuple $L^{\ell+1} = \langle H_0, H_1, \ldots, H_{k-1}, f\rangle$, where each H_h is an attention head and $f\colon ([2]^p)^k \times [2]^p \to [2]^p$ is a p-precision* activation *function.*

A p-precision transformer network layer can be understood to define a sequence of vectors $h_0^{\ell+1}, h_1^{\ell+1}, \ldots, h_{n-1}^{\ell+1}$ in terms of an input sequence of vectors $h_0^\ell, h_1^\ell, \ldots, h_{n-1}^\ell$ (coming from the previous layer in the transformer network) by first computing k attention heads in parallel and then combining their outputs using f. The first k inputs to f will correspond to the attention head outputs, and the additional input is the original input from the previous layer. Recall that $a_{i,h}^{\ell+1}$ is the output of head $H_{i,h}^{\ell+1}$ on input h^ℓ at position i. The function computed by a transformer network layer can be described formally as follows:

Definition 5.27. *For $\ell \geq 0$, a p-precision transformer network layer $L^{\ell+1}$ recurrently computes the output sequence $h_0^\ell, h_1^\ell, \ldots, h_{n-1}^\ell$ as a function of the inputs $h_0^\ell, h_1^\ell, \ldots, h_{n-1}^\ell$, where, for $1 \leq i \leq n$, the i-th component is computed according to*

$$h_i^{\ell+1} = f\big(a_{i,0}^{\ell+1}, a_{i,1}^{\ell+1}, \ldots, a_{i,k-1}^{\ell+1}, h_i^\ell\big).$$

This function f can be understood to encapsulate layer norm, residual connections, and the feedforward sub-layer of a standard transformer network. The component of the output sequence of the previous layer h_i^ℓ is given to f to allow residual connections. As mentioned previously, f can also encapsulate the value function for each head.

Finally, we can define a transformer network of depth d as a cascade of d transformer network layers:

Definition 5.28. *A p-precision transformer network over alphabet Σ is a pair consisting of a p-precision position embedding function $\varphi\colon \Sigma \times \mathbb{N} \to [2]^p$ and a d-tuple of p-precision transformer network layers $\big\langle L^1, L^2, \ldots, L^d\big\rangle$.*

For a position embedding function φ and $w \in \Sigma^n$, let $\varphi(w)$ be the position-wise broadcasted embedding of w: for $0 \leq i < n$, $\varphi_i(w) \coloneqq \varphi(w_i, i)$.

Definition 5.29. *A transformer network $\big(\varphi, \big\langle L^1, L^2, \ldots, L^d\big\rangle\big)$ computes the following function of a string $w \in \Sigma^*$:*

$$T(w) = \big(L^d \circ L^{d-1} \circ \cdots \circ L^1\big)(\varphi(w)).$$

We will use n to denote the length of w, and take the transformer network's depth d to be fixed with respect to n.

The *input* to the transformer network can thus be represented with $N = n \cdot \log(|\Sigma|)$ bits using a binary encoding for the vocabulary. The circuits we construct subsequently to simulate transformer networks will also have output size N. We will assume

transformer networks have log-precision relative to the size of the input, specifically $O(\log(N))$-precision. Since $|\Sigma|$ is fixed (typically 30000 in practice), we will think in terms of $O(\log(n))$-precision. Thus, by Definition 5.23, all the intermediate functions of such transformer networks are computable in $O(\log(n))$ space and output (at most) that many bits. Note that this is enough precision to represent positional encodings and for each position to point to a constant number of other values, but not enough precision for non-lossy pooling of the entire input into a single value.

Our log-precision transformer networks do not enforce that s and f follow the transformer network structure. However, a feedforward net whose primitive operations (for example scalar multiplication) are defined over $O(\log(n))$-size numbers can be computed in $O(\log(n))$ space. Thus, bounded-precision practical transformer networks are a special case of our log-precision transformer networks. This makes our setup appropriate for proving upper bound on transformer networks.

5.6.5 log-Precision Transformer Networks as nonuniform Threshold Circuits

We will first show that log-precision transformer networks can be simulated by *nonuniform* threshold circuits, before presenting the more technical *uniform* version of the results in subsection 5.6.6.

Corollary 5.30. *Let $f\colon [2]^* \to [2]^m$ be a function. For all $c \in \mathbb{R}^+$ and $n \in N$, there exists an AC^0 circuit of size at most $n^c + c \cdot \log(n) + m$ and depth 3 that computes f on inputs of size $c \cdot \log(n)$.*

Proof. This follows directly from Lemma 5.4. □

We now use Corollary 5.30 to prove the following nonuniform result. We note that the proof works even if the notion of p-precision is relaxed to not require computability in space p. This requirement will, however, become important for our subsequent result in subsection 5.6.6.

Theorem 5.31. *Any $c \cdot \log(n)$-precision depth-d transformer network operating on inputs in Σ^n can be simulated by a threshold circuit family of depth $3 + (9 + 2 \cdot d_{\oplus}) \cdot d$.*

Proof. Let $w \in \Sigma^n$ be the input of a $(c \cdot \log(n))$-precision transformer network. We show by induction that we can construct a composition of constant-depth, poly-size threshold circuits to compute each layer of this transformer network. Thus, any constant-depth transformer network will be computable by a constant-depth threshold circuit.

In the base case of layer 0 and token i, we construct gates representing the constant i encoded in binary. We can then compute $h_i^0 = \varphi(w_i, i)$ using Corollary 5.30, yielding a poly-size depth-3 circuit.

In the inductive case of computing layer $h_i^{\ell+1}$ for $0 \leq \ell < d$, we note that each vector output of layer h_i^ℓ has size (at most) $c \cdot \log(n)$ bits because of the log-precision assumption.

We first fix a head $a_{i,k}^{\ell+1}$ to simulate. Applying Corollary 5.30, we can compute $s\big(h_i^\ell, h_j^\ell\big)$ with a poly-size depth-3 circuit in parallel for all j. Since n floats with $c \cdot \log(n)$ precision can be approximately added in TC^0 [MS23b, Appendix A], we can construct a TC^0 circuit of depth $d_\oplus$ to compute Z_j. Since $s\big(h_i^\ell, h_j^\ell\big)$, Z_i, and h_i^ℓ all have $c \cdot \log(n)$ bits, we can compute $\frac{s\left(h_i^\ell, h_j^\ell\right)}{Z_i} \cdot h_j^\ell$ with a poly-size depth-3 circuit; we do this in parallel for all j. Next, we again use the fact that approximate addition of n floats is in TC^0 to compute $a_{i,h}^{\ell+1}$ as the approximate sum over j with a depth-$d_\oplus$ circuit.

We now simulate a layer $h_i^{\ell+1}$ in terms of its constituent heads. Since all arguments of g have size $c \cdot \log(n)$, we apply Corollary 5.30 to compute g with a poly-size depth-3 circuit, yielding $h_i^{\ell+1}$. We repeat this in parallel for all i. This completes the inductive step. The sub-circuit we have constructed for the $(\ell+1)$-st layer has a depth of $9+2 \cdot d_\oplus$.

Aggregating the circuit over all d layers, its overall depth is $3 + (9 + 2 \cdot d_\oplus) \cdot d$. □

The following now follows directly:

Corollary 5.32. *Any* log*-precision transformer network can be simulated by a nonuniform* TC^0 *circuit family.*

Proof. Since any given transformer network has some constant depth d, the depth of the resulting threshold circuit family is also constant. □

5.6.6 log-Precision Transformer Networks as Uniform Threshold Circuits

We will now extend the argument from the last section to show that $O(\log(n))$-precision transformer networks can be simulated by uniform constant-depth threshold circuits by capitalizing on the assumption that φ, s, and f are log-precision, and thus can be computed in $O(\log(n))$ space. The overall proof idea is similar, but due to the uniformity condition, the proof becomes substantially more technical. Not only must we show the existence of a threshold circuit family computing a transformer, but also that this circuit family can be generated by a log-space Turing machine.

We first extend Corollary 5.30 to respect uniformity:

Lemma 5.33. *Let* $f\colon [2]^* \to [2]^m$ *be a linear-space computable function. There exists a Turing machine that, for all* $n \in \mathbb{N}$ *and* $c \in \mathbb{R}^+$*, uses at most* $c \cdot \log(n) + \log(m)$ *space to map input* 1^n *to a circuit of size at most* $n^c + c \cdot \log(n) + m$ *and depth* 3 *that computes* f *on inputs of size at most* $c \cdot \log(n)$*.*

Proof. We give the proof in the form of an algorithm to construct a circuit as a function of n and then justify its correctness and space complexity.

Algorithm: We first print $2 \cdot c \cdot \log(n)$ nodes representing unnegated and negated input nodes.

Now, we need to show how to construct nodes corresponding to n^c DNF terms. To that end, we loop over all possible inputs $x \in [2]^{c \cdot \log(n)}$ by maintaining the $c \cdot \log(n)$ bit binary representation of x (initialized with $0^{c \cdot \log(n)}$) and incrementing it by 1 at each step of the loop. We create a new $\wedge$ node i with $c \cdot \log(n)$ arguments, defined as follows: For $j \in [c \cdot \log(n)]$, we create an argument pointer to (unnegated) node j if $x_j = 1$ and to (negated) node $c \cdot \log(n) + j$ otherwise.

Next, we construct nodes computing each of the m outputs. We loop over $k \in [m]$, constructing a single node for each k. We loop over all $x \in [2]^{c \cdot \log(n)}$ analogously above to construct a list of arguments. By our linear-space computability assumption and because x has $c \cdot \log(n)$ bits, we can compute $f(x)$ as a subroutine in $O(\log(n))$-space to obtain $f_k(x)$. If $f_k(x) = 1$, we print node $2 \cdot c \cdot \log(n) + j$ as an argument of node k.

Correctness: We show that this Turing machine M maps input 1^n to a serialized circuit computing f on inputs of size n. The first layer simply produces unnegated and negated input values. The second layer then produce all possible DNF terms. Finally, node k of the third layer computes the disjunction over all terms x such that $f_k(x) = 1$. Thus, node k of the third layer computes f_k.

Logarithmic Space: To complete the proof, we justify that M uses $O(\log(n) + \log(m))$ space. Looping over $x \in [2]^{c \cdot \log(n)}$ is accomplished by treating x as a binary number initialized to 0 and incrementing it at each step. Thus, the loop pointer for building the DNF terms takes $c \cdot \log(n)$ space to store. For building the m output nodes, we maintain a similar loop pointer as well as an index $k \leq m$, taking $c \cdot \log(n) + \log(m)$ space. Thus, the overall algorithm uses $c \cdot \log(n) + \log(m)$ space.

Thus, the Turing machine M uses $c \cdot \log(n) + \log(m)$ space to map 1^n to a circuit of size at most $n^c + c \cdot \log(n) + m$ and depth 3 that computes f on size $c \cdot \log(n)$ inputs. $\square$

We can leverage this lemma to derive the uniform analog of Theorem 5.31, as follows:

Theorem 5.34. *Any $c \cdot \log(n)$-precision depth-d transformer network operating on inputs in Σ^n can be simulated by a* log*-space-uniform threshold circuit family of depth $3 + (9 + 2 \cdot d_{\oplus}) \cdot d$.*

Proof. We will provide a proof by induction over the transformer network layers ℓ that there is a Turing machine M operating in $O(\log(n))$ space that, on input 1^n, outputs a circuit that simulates the transformer network's computation on inputs of size n. This

circuit is identical to the one in the proof of Theorem 5.31, and thus has the same circuit depth.

In the base case, we use logarithmic space to track a counter maintaining the current token i (between 1 and n) throughout the circuit construction. We construct gates encoding the constant i in binary. We can then apply Lemma 5.33 to construct a Turing machine that maps 1^n to a constant-depth threshold circuit computing $h_i^0 = \varphi(w_i, i)$.

In the inductive case, we assume we can output in $O(\log(n))$ space a circuit computing every value h_i^ℓ in the previous layer ℓ. We will show that we can, in $O(\log(n))$ space, now output a circuit computing every value in layer $\ell + 1$.

As in Theorem 5.31, we first fix a head $a_{i,h}^{\ell+1}$ to simulate. Recall that

$$a_{i,h}^{\ell+1} = \bigoplus_{j \in [n]} \frac{s\big(h_i^\ell, h_j^\ell\big)}{Z_i} \cdot h_j^\ell.$$

By Lemma 5.33, we can generate a depth-3 circuit of size at most $z = n^{c'} + c' \cdot \log(n) + 1$, where $c' = 2 \cdot c$ (since the input to f is of size $2 \cdot c \cdot \log(n)$), that computes $s(h_i^\ell, h_j^\ell)$ for specific i, j. We do this sequentially for $j \in [n]$ and $h \in [k]$, padding each circuit with unused nodes, such that each one has size exactly z, and the z-th node corresponds to the output. Thus, the indices of the output nodes for each of the columns will be $w_\ell + z \cdot (j \cdot k + h)$ for $j \in [n]$, where w_ℓ is the index of the last output node h_n^ℓ of the previous layer.

At this point, we use the fact that for $p = c \cdot \log(n)$, the p-precision approximate sum of n p-precision numbers can be computed by a uniform threshold circuit [MS23b, Appendix A]. We can thus use a Turing machine as a sub-routine to generate, on input 1^n, k threshold circuits, where each has size z' and computes a $\oplus$ gate over n items of precision p each. We set the inputs of circuit h to be nodes $w_\ell + z \cdot (j \cdot k + h)$ for $j \in [n]$. By construction, this yields the normalizing constants $Z_i = \bigoplus_{j \in [n]} s(h_i^\ell, h_j^\ell)$, whose value is located at the node at index $w_\ell + z \cdot n \cdot k + z'$ for head h.

Using p-precision arithmetic operator circuits, we can now also generate a circuit to compute $\frac{s(h_i^\ell, h_j^\ell)}{Z_i} \cdot h_j^\ell$ for each $j \in [n]$ and $h \in [k]$, by using index $w_\ell + z \cdot (j \cdot k + h)$ as before for the value of $s(h_i^\ell, h_j^\ell)$ and index $w\ell + z \cdot n \cdot k + z' \cdot h$ for the normalizing constant Z_i of head h. Here too we use circuits of identical size z'', making $w_\ell + k \cdot (z \cdot n + z' + z'' \cdot i)$ the index of the output nodes of these n circuits. Next, we again employ a $\oplus$ circuit of size z', similar to the computation of Z_i, to compute the sum of these n values. Finally, we compute $h_i^{\ell+1}$ by applying f via Lemma 5.33.

Note that this requires keeping only ℓ, i, and n in memory, each of which takes $O(\log(n))$ bits.

We repeat this process for all $i \in [n]$ to compute the entire layer $\ell + 1$, which finishes

the inductive step: If we can output a circuit computing layer ℓ in $O(\log(n))$ space, then we can do the same for layer $\ell + 1$. □

Because the depth derived in Theorem 5.34 is constant with respect to n, it follows that:

Corollary 5.35. *Any* log*-precision (or fixed-precision) transformer network can be simulated by a uniform* TC^0 *circuit family.*

We can now use this result to establish a connection to the logic FO(M) we defined in section 4.2 [MS23a, Theorem 2].

Corollary 5.36. *The output of any* log*-precision transformer network can be expressed in* FO(M)*.*

Proof. This follows directly from the equivalence of log-uniform TC^0 and FO(M) which has been shown in [MIS90, section 9]. □

For fixed-precision transformer networks using soft attention, we can even combine this result with the results from section 5.5 to show that they are strictly less powerful than TC^0 circuits.

Corollary 5.37. *The class of languages recognizable by a fixed-precision soft-attention transformer network is a proper subset of uniform* TC^0*.*

Proof. From Corollary 5.35, we know that it is a subset. If we now assume that the class is equal to uniform TC^0, then it follows from Theorem 5.20 that $\mathrm{TC}^0 \subseteq \mathrm{FOC}[+; \mathrm{MOD}]$, which contradicts Theorem 5.21. Since our assumption leads to a contradiction, it has to be wrong and the subset relation is thereby proper. □

5.7 Lower Bounds for Instruction Following and Advice Transformers

This section is also based on [MS23b] and the definitions from section 5.6 still apply here.

5.7.1 Circuit Value Prolem

So far, we have shown that log-uniform TC^0 is an upper bound for log-precision transformer networks. Is this upper bound tight, that is, also a lower bound? While we do not answer this question here, we address a related question as a first step: We construct a transformer network that can evaluate TC^0 circuits on binary inputs, showing

that transformers can compute any TC^0 function when their input is augmented with the right "instructions".

More formally, we consider the Circuit Value Problem (CVP) [Lad75, p. 18], also referred to as the Circuit Evaluation Problem, where the input is a boolean circuit C and a string $x \in [2]^n$, and the task is to return the value of $C(x) \in [2]$. This problem is known to be complete for the class P under log-space reduction [Lad75, p. 19]. We will assume C is serialized as described in subsection 5.6.1 and prove that log-precision transformer networks can evaluate any TC^0 circuit. Note that this is an extension of the typical CVP since the circuit has threshold gates, not just standard AND/OR gates.

To demonstrate the practicality of this lower bound construction, we will not just prove the existence of transformers that can evaluate TC^0 circuits but also specify concrete choices for the positional embedding scheme and the class of attention functions that are sufficient to do so.

Fractional Positional Embeddings: For a vector x and scalar y, let $\langle x, y \rangle$ be the vector obtained by appending y onto x. For $\sigma \in \Sigma$, let $v(\sigma)$ be the one-hot embedding of σ into $\mathbb{R}^{|\Sigma|}$. For $w \in \Sigma^*$ and $i \in \mathbb{N}$, the fractional positional embedding at token i is

$$\varphi(w_i, i) = \left\langle v(w_i), \frac{i}{n} \right\rangle.$$

Saturated Attention: We imagine $f(h_i^\ell, h_j^\ell)$ is computed via saturated attention, which provides a simple model of the types of attention we can expect to be learned in transformers. First, queries are computed as $q_i = Qh_i^\ell$, and then keys $k_j = Kh_j^\ell$. Define the dot product attention score $\sigma_{i,j} = q_i^\mathsf{T} k_j$. We can then define saturated attention as

$$s\big(h_i^\ell, h_j^\ell\big) := \begin{cases} 1, & \text{if } \sigma_{i,j} = \max_k \sigma_{i,k} \\ 0, & \text{otherwise.} \end{cases}$$

After normalization, saturated attention creates a distribution that is uniform over a subset of positions. Thus, it is capable of parameterizing hard attention, uniform attention over the full sequence, and various attention patterns in between.

Simple Pooling Functions: For simplicity, we assume pooling functions f are thresholded linear functions of their inputs. Thus, they could be implemented by a feedforward neural net. Without loss of generality, we let attention heads have a value function, which can be folded into the pooling function from the last layer (see subsection 5.6.4).

Terminology: We use the term input node to mean a token of type X and gate node to mean a token of type Dir. We call a token of type & an argument.

We are now ready to present the result. Our construction below is specific to circuits serialized in prefix form (see subsection 5.6.1), but it can be extended to other serializations as well.

Lemma 5.38. *For all $d \in \mathbb{N}$, there exists a transformer network with fractional positional embeddings, saturated attention, thresholded linear pooling functions, and depth $2 \cdot d$ that, for any threshold circuit C of depth d serialized in prefix form, maps input $\langle C, x \rangle$ to the value $C(x)$.*

Proof. We will construct a pair of two transformer network layers that evaluate all the nodes at depth ℓ in the threshold circuit, for any ℓ. It follows that a transformer network of depth $2 \cdot d$ can compute the value $C(x)$.

Base Case: Input Nodes. We use an attention layer to attend uniformly over all positions with value 1 if $w_i = \mathsf{X}$ and 0 otherwise. This head computes $\frac{|w|_{\mathsf{X}}}{n}$, where $|w|_{\mathsf{X}}$ is the number of occurrences of X in w. A second layer, then, at input node i, computes the positional embedding of the token representing input value x_i:

$$\frac{1 - |w|_{\mathsf{X}} + i}{n}.$$

We attend to this position to retrieve x_i. After these layers, each input node i stores its value x_i. We also use the base-case layers to construct an attention head that, at the i-th node, counts the fraction of tokens (out of n) that are nodes to the left of the current node. Thus, the column corresponding to node i stores the value $\frac{i}{n}$.

At each gate node i, we use two more attention heads to find the index of the next & to the right, and then count the fraction of tokens before it that are 1. This head thus computes $\frac{k_i}{m_i}$ where k_i is the threshold value of gate i and m_i is its arity.

Finally, using the first attention layer, we have each 1 node attend to the first argument symbol & to its left and retrieve its index $\frac{p}{n}$. Then, in the second attention layer, each argument attends uniformly over all nodes with values $\frac{p}{n}$. The net effect is for each argument to store $\frac{j}{n}$, that is, the pointer it is encoding in unary as $\&1^j$.

Inductive Case: Gate Nodes. By our inductive assumption over prior layers, all tokens corresponding to circuit nodes at depth $\leq \ell$ contain their appropriate value. We now construct 2 transformer network layers to evaluate gate nodes at depth $\ell + 1$.

In the first attention layer, each argument token attends to the closest gate node i to its left, which is the gate it belongs to. Recall from the base case that argument token & already stores $\frac{j}{n}$, where j is the pointer value it encodes. Each argument token now attends with query $\frac{j}{n}$ to retrieve from node j its already computed value.

The second attention layer applies at gate nodes, not arguments. At gate i of arity m_i, we set the attention $s(i, j)$ to indicate whether argument j belongs to gate node i,

which holds for exactly m_i arguments. We set the attention value at argument j to be the binary value of node j, which was retrieved in the previous paragraph. Thus, the attention head computes $\frac{c_i}{m_i}$, where c_i is the number of arguments of node i that are 1. We repeat this for all gate nodes.

At this point, we have both the count of true inputs to gate node i, $\frac{c_i}{m_i}$, and, from the base case, the threshold parameter of gate i, $\frac{k_i}{m_i}$. Thresholding $\frac{c_i - k_i}{m_i}$ at 0 allows us to decide, based on whether Dir is <= or >=, whether the current gate node should output a 0 or a 1. Repeating this for all gates at layer $\ell + 1$ completes the inductive step: We can evaluate all gate nodes in this layer. □

The following is an immediate consequence of this:

Theorem 5.39. *Depth-$(2 \cdot d)$ transformer networks can solve CVP for depth-d TC^0 circuits.*

Proof. According to Lemma 5.38, there is a transformer network for any circuit depth d that solves the problem. □

5.7.2 Instruction Following

CVP is closely related to instruction learning and instruction following tasks. The latter task setup provides a transformer network two inputs: a regular expression r as an "instruction", and $z \in [2]^*$. The goal of the task is to return whether z belongs to the regular language represented by r. Viewed from this lens, the circuit evaluation setup asks: Can transformers follow instructions provided in the form of a circuit? We will show that the answer is yes for all constant-depth threshold circuits.

Formally, an instruction I is any description of a function f_I of $[2]^*$, that is, a fixed-size program to compute that function under some model of computation. We say a transformer network correctly follows an instruction I iff, for all $x \in [2]^*$, it correctly computes $f_I(x)$ on input $\langle I, x \rangle$. A nonuniform instruction description is a family of length-specific descriptions $(I_n)_{n \in \mathbb{N}}$. We say a transformer network correctly follows a nonuniform instruction family I_n iff, for all $n \in \mathbb{N}$ and all $x \in [2]^n$, it correctly computes $f_I(x)$ on input $\langle I_n, x \rangle$. The nonuniform description I_n may take any form. When it forms a TC^0 circuit family, we refer to it as a TC^0 instruction description.

Corollary 5.40. *There exists a depth-$(2 \cdot d)$ transformer network that can correctly follow any depth-d TC^0 instruction description.*

Proof. This follows, since Lemma 5.38 constructs a transformer network that can evaluate any TC^0 circuit. □

Thus, transformer networks with simple position embeddings, attention, and pooling functions can simulate any instruction provided in the form of a TC^0 circuit. We note that, while it is unknown whether the class of regular languages is contained in TC^0, the other direction is known: There are problems computable by TC^0 circuits that are not regular. These include problems involving counting and arithmetic, which are beyond regular languages. These results thus expand the known kinds of instructions transformers are able to follow, at least with hand-constructed weights.

5.7.3 Advice Transformers

We can also view circuit evaluation abilities of transformers (Lemma 5.38) from the lens of advice-taking Turing machines which, in addition to their usual input, are also provided an input-length-dependent (but input-independent) advice string. For instance, P/poly is the class of problems decidable in polynomial time when the Turing machine is given an advice string of size polynomial in the input length.

In the same vein, let T/poly be the class of log-precision, constant-depth transformer networks with polynomial advice strings. In other words, on an input of size n, we allow the transformer network to receive an additional $\text{poly}(n)$ bits of input that cannot depend on the standard input. Now we can show:

Corollary 5.41. $TC^0 \subseteq$ T/poly.

Proof. Let $(C_n)_{n\in\mathbb{N}}$ be a circuit family demonstrating that a problem is in nonuniform TC^0. Then, by passing the description of C_n as advice for input length n, it immediately follows from Lemma 5.38 that advice transformer networks can simulate nonuniform TC^0. □

Since non-uniform TC^0 even contains some undecidable languages, T/poly is clearly a very powerful class. Thus, a problem in T/poly cannot always be solved by a transformer network on its own. However, if given a description of how to do so ("advice") in the form of a TC^0 circuit, we have shown that a transformer network could solve that problem.

6 Conclusion

6.1 Interpretation

At first, we have seen choosing an overly simplified attention function, namely hard attention, for our model of transformer networks bounds them in AC^0. This does not reflect the capabilities of transformer networks in the real world, as they have been shown to have the ability to count, which AC^0 circuits are not capable of [BAG20, section 6; MSS22, Section 1]. Then, we have seen that assuming arbitrary precision in our model once again leads to results that do not reflect the capabilities of actual transformer networks, such as being able to recognize any formal language and being Turing complete.

For the much more realistic model of log-precision transformer networks, we have shown that they can be simulated by log-uniform TC^0 circuits, for any kind of attention function. This establishes threshold functions as a fundamental operation for understanding the computational model of transformers. This result also establishes potential limits on the computational power of log-precision transformer networks. For example, if $L \subsetneq P$, transformer networks cannot compute all polynomial time functions. They are certainly very far from being universal. The intuition at the heart of this result is that forcing a model to be highly parallelizable likely sacrifices its expressiveness. Since parallelism seems essential to pretraining any massive model at scale, any large language model — transformer network or otherwise — may suffer from a similar tradeoff [MS23b, section 8].

6.2 Future Outlook

Most of the research into the expressivity of transformer networks has, thus far, gone into establishing upper bounds, that is, finding out what kinds of problems they cannot solve. A next step would be to figure out corresponding lower bounds to more closely understand what exactly these networks are capable of.

Another area for further research would be to search for a logic or some kind of specialized programming language that is equivalent to transformer networks. This could allow translating the internals of a transformer network into something more

human-readable. It could also allow a new angle for a theoretical analysis.

But transformer networks are of course not the only deep neural networks. The field is evolving rapidly, and there is also much to be learned from the analysis of competing architectures, such as the Mamba network [GD23]. Instead of focusing on individual architectures, it might be of interest to investigate whether the parallelism tradeoff is real and what that would imply for future design of large language models.

Bibliography

[BAG20] Bhattamishra, Satwik ; Ahuja, Kabir ; Goyal, Navin: On the Ability and Limitations of Transformers to Recognize Formal Languages. In: Webber, Bonnie (Hrsg.) ; Cohn, Trevor (Hrsg.) ; He, Yulan (Hrsg.) ; Liu, Yang (Hrsg.): *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing, EMNLP 2020, Online, November 16-20, 2020*, Association for Computational Linguistics, 2020, 7096–7116

[BCB15] Bahdanau, Dzmitry ; Cho, Kyunghyun ; Bengio, Yoshua: Neural Machine Translation by Jointly Learning to Align and Translate. In: Bengio, Yoshua (Hrsg.) ; LeCun, Yann (Hrsg.): *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, 2015

[BJZP20] Basodi, Sunitha ; Ji, Chunyan ; Zhang, Haiping ; Pan, Yi: Gradient amplification: An efficient way to train deep neural networks. In: *Big Data Min. Anal.* 3 (2020), Nr. 3, 196–207. http://dx.doi.org/10.26599/BDMA.2020.9020004. – DOI 10.26599/BDMA.2020.9020004

[BMR+20] Brown, Tom B. ; Mann, Benjamin ; Ryder, Nick ; Subbiah, Melanie ; Kaplan, Jared ; Dhariwal, Prafulla ; Neelakantan, Arvind ; Shyam, Pranav ; Sastry, Girish ; Askell, Amanda ; Agarwal, Sandhini ; Herbert-Voss, Ariel ; Krueger, Gretchen ; Henighan, Tom ; Child, Rewon ; Ramesh, Aditya ; Ziegler, Daniel M. ; Wu, Jeffrey ; Winter, Clemens ; Hesse, Christopher ; Chen, Mark ; Sigler, Eric ; Litwin, Mateusz ; Gray, Scott ; Chess, Benjamin ; Clark, Jack ; Berner, Christopher ; McCandlish, Sam ; Radford, Alec ; Sutskever, Ilya ; Amodei, Dario: Language Models are Few-Shot Learners. In: Larochelle, Hugo (Hrsg.) ; Ranzato, Marc'Aurelio (Hrsg.) ; Hadsell, Raia (Hrsg.) ; Balcan, Maria-Florina (Hrsg.) ; Lin, Hsuan-Tien (Hrsg.): *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*, 2020

[CCP23] Chiang, David ; Cholak, Peter ; Pillay, Anand: Tighter Bounds on the Expressivity of Transformer Encoders. In: Krause, Andreas (Hrsg.) ; Brunskill, Emma (Hrsg.) ; Cho, Kyunghyun (Hrsg.) ; Engelhardt, Barbara (Hrsg.) ; Sabato, Sivan (Hrsg.) ; Scarlett, Jonathan (Hrsg.): *International Conference on Machine Learning, ICML 2023, 23-29 July 2023, Honolulu, Hawaii, USA* Bd. 202, PMLR, 2023 (Proceedings of Machine Learning Research), 5544–5562

[DCLT19] Devlin, Jacob ; Chang, Ming-Wei ; Lee, Kenton ; Toutanova, Kristina: BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. In: Burstein, Jill (Hrsg.) ; Doran, Christy (Hrsg.) ; Solorio, Thamar (Hrsg.): *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT 2019, Minneapolis, MN, USA, June 2-7, 2019, Volume 1 (Long and Short Papers)*, Association for Computational Linguistics, 2019, 4171–4186

[GD23] Gu, Albert ; Dao, Tri: Mamba: Linear-Time Sequence Modeling with Selective State Spaces. In: *CoRR* abs/2312.00752 (2023). http://dx.doi.org/10.48550/ARXIV.2312.00752. – DOI 10.48550/ARXIV.2312.00752

[Gol73] Goldstein, L. J.: A History of the Prime Number Theorem. In: *The American Mathematical Monthly* 80 (1973), Nr. 6, 599-615. http://dx.doi.org/10.1080/00029890.1973.11993338. – DOI 10.1080/00029890.1973.11993338

[HAF22] Hao, Yiding ; Angluin, Dana ; Frank, Robert: Formal Language Recognition by Hard Attention Transformers: Perspectives from Circuit Complexity. In: *Trans. Assoc. Comput. Linguistics* 10 (2022), 800–810. https://transacl.org/ojs/index.php/tacl/article/view/3765

[Lad75] Ladner, Richard E.: The circuit value problem is log space complete for *P*. In: *SIGACT News* 7 (1975), Nr. 1, 18–20. http://dx.doi.org/10.1145/990518.990519. – DOI 10.1145/990518.990519

[MIS90] Mix Barrington, David A. ; Immerman, Neil ; Straubing, Howard: On Uniformity within NC^1. In: *J. Comput. Syst. Sci.* 41 (1990), Nr. 3, 274–306. http://dx.doi.org/10.1016/0022-0000(90)90022-D. – DOI 10.1016/0022–0000(90)90022–D

[MS23a] Merrill, William ; Sabharwal, Ashish: A Logic for Expressing Log-Precision Transformers. In: Oh, Alice (Hrsg.) ; Naumann, Tristan (Hrsg.)

; Globerson, Amir (Hrsg.) ; Saenko, Kate (Hrsg.) ; Hardt, Moritz (Hrsg.) ; Levine, Sergey (Hrsg.): *Advances in Neural Information Processing Systems 36: Annual Conference on Neural Information Processing Systems 2023, NeurIPS 2023, New Orleans, LA, USA, December 10 - 16, 2023*, 2023

[MS23b] Merrill, William ; Sabharwal, Ashish: The Parallelism Tradeoff: Limitations of Log-Precision Transformers. In: *Transactions of the Association for Computational Linguistics* 11 (2023), 06, 531-545. http://dx.doi.org/10.1162/tacl_a_00562. – DOI 10.1162/tacl_a_00562. – ISSN 2307–387X

[MSS22] Merrill, William ; Sabharwal, Ashish ; Smith, Noah A.: Saturated Transformers are Constant-Depth Threshold Circuits. In: *Trans. Assoc. Comput. Linguistics* 10 (2022), 843–856. https://transacl.org/ojs/index.php/tacl/article/view/3465

[Ope23] OpenAI: GPT-4 Technical Report. In: *CoRR* abs/2303.08774 (2023). http://dx.doi.org/10.48550/ARXIV.2303.08774. – DOI 10.48550/ARXIV.2303.08774

[PBM21] Pérez, Jorge ; Barceló, Pablo ; Marinkovic, Javier: Attention is Turing-Complete. In: *J. Mach. Learn. Res.* 22 (2021), 75:1–75:35. http://jmlr.org/papers/v22/20-302.html

[PMB19] Pérez, Jorge ; Marinkovic, Javier ; Barceló, Pablo: On the Turing Completeness of Modern Neural Network Architectures. In: *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*, OpenReview.net, 2019

[PW17] Press, Ofir ; Wolf, Lior: Using the Output Embedding to Improve Language Models. In: Lapata, Mirella (Hrsg.) ; Blunsom, Phil (Hrsg.) ; Koller, Alexander (Hrsg.): *Proceedings of the 15th Conference of the European Chapter of the Association for Computational Linguistics, EACL 2017, Valencia, Spain, April 3-7, 2017, Volume 2: Short Papers*, Association for Computational Linguistics, 2017, 157–163

[RNSS18] Radford, Alec ; Narasimhan, Karthik ; Salimansm, Tim ; Sutskever, Ilya: Improving Language Understanding by Generative Pre-Training. (2018)

[RWC+19] Radford, Alec ; Wu, Jeff ; Child, Rewon ; Luan, David ; Amodei, Dario ; Sutskever, Ilya: Language Models are Unsupervised Multitask Learners. (2019)

[Str23] Strobl, Lena: Average-Hard Attention Transformers are Constant-Depth Uniform Threshold Circuits. In: *CoRR* abs/2308.03212 (2023). http://dx.doi.org/10.48550/ARXIV.2308.03212. – DOI 10.48550/ARXIV.2308.03212

[Vol99] Vollmer, Heribert: *Introduction to Circuit Complexity - A Uniform Approach*. Springer, 1999 (Texts in Theoretical Computer Science. An EATCS Series). http://dx.doi.org/10.1007/978-3-662-03927-4. http://dx.doi.org/10.1007/978-3-662-03927-4. – ISBN 978–3–540–64310–4

[VSP+17] Vaswani, Ashish ; Shazeer, Noam ; Parmar, Niki ; Uszkoreit, Jakob ; Jones, Llion ; Gomez, Aidan N. ; Kaiser, Lukasz ; Polosukhin, Illia: Attention is All you Need. In: Guyon, Isabelle (Hrsg.) ; Luxburg, Ulrike von (Hrsg.) ; Bengio, Samy (Hrsg.) ; Wallach, Hanna M. (Hrsg.) ; Fergus, Rob (Hrsg.) ; Vishwanathan, S. V. N. (Hrsg.) ; Garnett, Roman (Hrsg.): *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, December 4-9, 2017, Long Beach, CA, USA*, 2017, 5998–6008