ESSLLI 2024 Course Notes

Expressivity of Transformers:
Logic, Circuits, and Formal Languages

David Chiang
University of Notre Dame
dchiang@nd.edu

Jon Rawski
MIT/San Jose State University
jon.rawski@sjsu.edu

Lena Strobl
Umeå University
lena.strobl@umu.se

Andy Yang
University of Notre Dame
ayang4@nd.edu

Contents

1 Introduction **1**
1.1 Overview 1
1.2 Preliminaries 2
1.2.1 Linear algebra 2
1.2.2 Strings 2
1.2.3 Miscellaneous 3
1.3 Formal language and complexity theory 3
1.3.1 Star-free regular languages 4
1.3.2 Circuit complexity 9
1.4 Transformers 11
1.4.1 Input layer 11
1.4.2 Attention 12
1.4.3 Feed-forward networks 13
1.4.4 Layer normalization 15
1.4.5 Hidden layers 16
1.4.6 Transformer encoders 16
1.4.7 Remarks 17

2 Encoders with Unique-Hard Attention **18**
2.1 Unique-Hard Attention 18
2.2 Overview 18
2.3 Main Result 19
2.4 Additional Results 24
2.4.1 Stutter-Invariance 24
2.4.2 Regular Languages in AC^0 25
2.4.3 Depth 26

3 Decoders with Intermediate Steps **28**
3.1 Definitions 28
3.1.1 Average-hard attention 28
3.1.2 Transformer decoders 28
3.1.3 Intermediate steps 29
3.1.4 Turing machines 30
3.2 Transformers Simulating Turing Machines 30
3.3 Open Questions 35

4 Encoders with Soft Attention **36**
4.1 Upper Bound 36
4.1.1 Arithmetic predicates 36
4.1.2 Uniform TC^0 37
4.1.3 Counting quantifiers 38
4.1.4 Precision 39
4.1.5 Main result 40
4.2 Lower bound 41
4.2.1 $\mathsf{K}_t[\#,+]$ 41
4.2.2 Boolean and Count Representations 42
4.2.3 Counting 43
4.2.4 Linear constraints 44
4.2.5 Main Result 45

Chapter 1

Introduction

1.1 Overview

This course is a survey of current knowledge about expressivity of transformers from the point of view of formal languages.

Transformers [Vaswani et al., 2017] are the neural network architecture underlying nearly every state-of-the-art model in natural language processing tasks, and have been extended to other fields as well. We will describe them in more detail in Section 1.4.

Expressivity studies the abilities and limitations: what class of problems can and can't be solved intrinsically by a particular class of models (here, transformers)? This differs from *learnability*, which concerns what problems models can or can't be trained to solve from data instances. Learnability is probably a more fundamental question, but expressivity is a prerequisite for learnability and the focus of this course. This is because there is No Free lunch: A model which successfully learns/induces some class of tasks pays for it by failing to learn other classes of tasks [Wolpert, 2021]. In short, a model's architecture determines its learnability, and knowledge is a prerequisite of learning.

The two seminal papers in this area are often summarized as follows:

> Transformers are Turing-complete [Pérez et al., 2019, Pérez et al., 2021].

> Transformers, given a string of `0`'s and `1`'s, cannot tell whether the number of `1`'s is odd or even [Hahn, 2020].

How can both of these statements be true? They rely on different assumptions about what a transformer is and what it means for a transformer to recognize a formal language. One of the goals of this course is to disentangle these assumptions and help you to understand how all the results in this area of research fit together.

The general picture that emerges has three parts (which correspond to days 2–4 of this course).

The claim that transformers cannot recognize PARITY (whether the number of `1`'s is odd or even) is only true for transformers with *unique-hard attention*, which are simpler than the transformers used in practice. Continued investigation of these transformers has led to exact characterizations: for instance, transformers with unique-hard attention and *strict future-masking* is exactly equivalent to the *star-free* languages.

The claim that transformers are Turing-complete relies crucially on the assumption that a transformer can take any number of *intermediate steps* before generating a final answer. This theoretical finding anticipated, by several years, the empirical finding that transformer language models reason better when encouraged to generate a *scratchpad* [Nye et al., 2022] or *chain of thought* [Wei et al., 2022].

Finally, for transformers that use soft-attention (as real transformers do), we don't have any exact characterizations yet. An upper bound is $\mathsf{DLOGTIME}$-uniform TC^0 or $\mathsf{FOM}[\mathrm{BIT}]$. Assuming the widely-believed conjecture that $\mathsf{TC}^0 \neq \mathsf{NC}^1$, this excludes some natural-looking problems like Boolean formula evaluation (that is, accepting true formulas like $(1 \vee 0 \vee 0) \wedge (0 \vee 1 \vee 1)$). Lower bounds are various extensions of first-order logic or linear temporal logics that add counting operators. This includes languages like the Dyck language. But the gap between these upper and lower bounds still seems rather wide, and narrowing this gap is an area of ongoing research.

Understanding the expressivity of transformers is crucial for both theory and practice. Theoretically, it helps us identify the limits of their capabilities, avoiding costly and tiresome experimentation. Practically, it informs the design of more effective models and algorithms, optimizing their performance for specific tasks in natural language processing and beyond.

1.2 Preliminaries

We write $\mathbb{N}$ for the set of natural numbers including 0, and $\mathbb{N}_{>0}$ for the natural numbers excluding 0. We write $[n]$ for the set $\{0, \ldots, n-1\}$ (note: not $\{1, \ldots, n\}$).

We write $\log x$ for the natural logarithm of x and $\log_2 x$ for the base-2 logarithm of x. In $O(\log n)$, the base does not matter, so we omit it. We write either e^x or $\exp x$ for the exponential function and sometimes we write $\exp_2 x$ for 2^x. We write $O(\mathrm{poly}(n)) = \bigcup_{k \geq 0} O(n^k)$.

1.2.1 Linear algebra

Variables that stand for vectors are lowercase boldface letters: $\mathbf{a}, \mathbf{b}, \ldots$. We write $\mathbf{0}$ for the zero vector. Variables that stand for matrices are uppercase boldface letters: $\mathbf{A}, \mathbf{B}, \ldots$.

If $\mathbf{x} \in \mathbb{R}^d$, we will normally write the components of $\mathbf{x}$ as $x_0, \ldots, x_{d-1}$ (note: 0-based indexing). But sometimes this is not convenient, so we also use a more code-like notation where the components of $\mathbf{x}$ are $\mathbf{x}[0], \ldots, \mathbf{x}[d-1]$. If we write $\mathbf{x}_0, \mathbf{x}_1$, these are names of two different vectors, and similarly for $\mathbf{x}^{(0)}, \mathbf{x}^{(1)}$. So, subscript $_i$ and superscript $^{(i)}$ mean "the i-th thing," but bracketed $[i]$ means "the i-th element of."

For $i \in [d]$, we write $e(i)$ for the i-th unit vector of the standard basis of $\mathbb{R}^d$, that is, the vector with a 1 in the i-th component and 0 everywhere else.

1.2.2 Strings

If A is any set, we write A^* for the set of finite sequences of elements A. If Σ is a finite alphabet (set of symbols), then Σ^* is called the set of strings over A. We also often deal with sequences of vectors; we write (nonstandardly) $(\mathbb{R}^d)^*$ for the set of finite sequences of vectors in $\mathbb{R}^d$.

If $\mathbf{a} \in A^*$, we write $|\mathbf{a}|$ for the length of $\mathbf{a}$ and a_i or $\mathbf{a}[i]$ for the i-th element of $\mathbf{a}$. As with vectors, we number the elements starting from 0, not 1. If $\mathbf{a}, \mathbf{b} \in A^*$, we write $\mathbf{ab}$, $\mathbf{a} \circ \mathbf{b}$, or sometimes $\mathbf{a} \cdot \mathbf{b}$ for the concatenation of $\mathbf{a}$ and $\mathbf{b}$.

A function from A^* to B^* is *length-preserving* if for all $\mathbf{a} \in A^*$, we have $|\mathbf{a}| = |f(\mathbf{a})|$. In this case, we write $f : A^* \xrightarrow{\text{lp}} B^*$.

1.2.3 Miscellaneous

Iverson bracket For any true or false statement ϕ, we write

$$\mathbb{I}[\phi] = \begin{cases} 1 & \text{if } \phi \text{ is true} \\ 0 & \text{if } \phi \text{ is false.} \end{cases} \tag{1.1}$$

Dot notation We use the following "dot notation," unusual in mathematics but common in programming languages. Whenever we define a neural network or a piece of a neural network as a function f, any variable x appearing in the definition of f can be referred to later using the notation $f.x$. For example, if $f : \mathbb{R}^d \to \mathbb{R}^d$ then we may refer to the dimension of its input/output space as $f.d$.

Exercise 1.1. Did you catch all of the following idiosyncratic notations?

1. Does $[n]$ start with 0 or 1? Does it end with $n - 1$ or n?
2. If $\mathbf{x}$ is a vector (or string), are its elements numbered starting from 0 or 1?
3. If $\mathbf{x}$ is a vector (or string), is its first element $\mathbf{x}_0$ or $\mathbf{x}[0]$?
4. What does $(\mathbb{R}^d)^*$ mean?
5. What does $f : A^* \xrightarrow{\text{lp}} B^*$ mean?
6. What does $f.x$ mean?

1.3 Formal language and complexity theory

In this section, we'll learn about a subclass of regular languages called the star-free regular languages (Section 1.3.1), which can be characterized by subclasses of finite automata and regular expressions, as well as by two logics, first-order logic and linear temporal logic. In the following section, we'll learn about a new model of computation, Boolean circuits (Section 1.3.2).

Why so many different formalisms? Much like a camera, each provides a different "lens" into aspects of computation:

- Automata provide insight into sequential processing of strings and, by extension, programs.
- Logical formulas are more human-readable (especially for ESSLLI participants, presumably) and logics provide very fine control over computational resources.
- Circuits have the clearest connection with neural networks, which are essentially arithmetic circuits with some extra operations. They provide insight into parallel processing of strings.

1.3.1 Star-free regular languages

Star-free languages are named for their characterization by star-free regular expressions. While we won't use star-free regular expressions later, understanding the origin of the name provides valuable intuition.

Definition 1.2 (star-free regular language)**.** The star-free languages over a finite alphabet Σ are exactly those that can be constructed using concatenation, union and complement from the empty language, $\{\epsilon\}$, and Σ. That is, they are the languages of star-free regular expressions, defined in BNF as:

$$\alpha ::= \emptyset \mid \epsilon \mid \sigma \mid \alpha_1 \cup \alpha_2 \mid \alpha_1\alpha_2 \mid \alpha^{\mathsf{C}} \tag{1.2}$$

where $\sigma \in \Sigma$.

Example 1.3. Let $\Sigma = \{\mathtt{a}, \mathtt{b}\}$.

- Σ^* is star-free because $\Sigma^* = \emptyset^{\mathsf{C}}$.
- $(\mathtt{ab})^*$ is star-free because $(\mathtt{ab})^* = (\mathtt{b}\Sigma^* \cup \Sigma^*\mathtt{a} \cup \Sigma^*\mathtt{aa}\Sigma^* \cup \Sigma^*\mathtt{bb}\Sigma^*)^{\mathsf{C}}$.
- $(\mathtt{aa})^*$ is regular but not star-free.

For more on star-free languages, see the survey by Pin [2020], and for their history, see Straubing [2018].

Counter-free automata

In the decades after these discoveries, researchers noticed deep equivalences between star-free languages and behaviors that were considered "counter-free":

Definition 1.4 (counter-free language)**.** A regular language is *counter-free* iff there exists some $n > 0$ (depending only on the language) such that for all strings $\mathbf{u}, \mathbf{v}, \mathbf{w} \in \Sigma^*$, where $|\mathbf{v}| \geq 1$, and for all $i \geq 1$

$$\mathbf{u}\mathbf{v}^n\mathbf{w} \in L \Leftrightarrow \mathbf{u}\mathbf{v}^{n+i}\mathbf{w} \in L.$$

Jäger and Rogers [2012] interpret this definition as a "cognitive" characterization of star-freeness:

> "Any cognitive mechanism that can distinguish member strings from non-members of a Star-Free language must be sensitive, at least, to the order of the length k blocks of symbols, for some fixed k and some fixed maximum length of the sequences of blocks, that occur in the presentation of the string.
>
> If the strings are presented as sequences of symbols in time, then this corresponds to being sensitive to the set of sequences, up to that maximum length, of the length k blocks that have occurred at any prior point.
>
> Any cognitive mechanism that is sensitive only to the set of fixed length sequences of length k blocks of symbols in the presentation of a string will be able to recognize *only* Star-Free languages."

Such a characterization holds for any mechanism, human or neural network. Many linguistic phenomena, including virtually all phonology and morphology, are characterized by Star-Free Languages [Heinz, 2018]. Let's look at a star-free linguistic example given at ESSLLI 2010 by Jim Rogers: a (simplified) syntactic phenomenon called possessor recursion. The main idea is that, if having n possessors is grammatical, $n+i$ must also be grammatical. In English, we can create pairs as below:

Example 1.5 (English possessor recursion)**.**

my mother's mother's mother resembled my mother	$\in L$
my mother's mother's $\underbrace{\text{(mother's)}}_{\geq 1}$ mother resembled my mother	$\in L$

Now let's imagine an alien language where possessor recursion is constrained to only even numbers. So in this case, the following intuitions would be expected:

Example 1.6 (Martian possessor recursion)**.**

my mother's mother's mother resembled my mother	$\in L$
my mother's mother's mother's mother resembled my mother	$\notin L$

This additional constraint violates non-counting behavior, and consequently is not star-free (but is still Regular).

We can also apply this property of counter-freeness directly to DFAs:

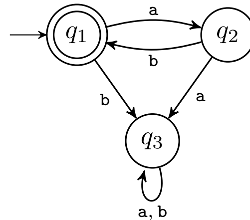
Definition 1.7 (counter-free automaton)**.** Let $M = (Q, \Sigma, \delta, q_0, F)$ be a DFA. Define the relation $q \xrightarrow{\mathbf{w}} r$, where $q, r \in Q$ and $\mathbf{w} \in \Sigma^*$, to mean "If M is in state q and reads string $\mathbf{w}$, then it ends up in state q." That is:

- $q \xrightarrow{\epsilon} q$ (where $q \in Q$).
- $q \xrightarrow{\sigma\mathbf{v}} s$ (where $q, s \in Q$, $\sigma \in \Sigma$, and $\mathbf{v} \in \Sigma^*$) iff, for some $r \in Q$, we have $\delta(q, \sigma) = r$ and $r \xrightarrow{\mathbf{v}} s$.

We say that M is *counter-free* iff there is an N such that for all $\mathbf{w} \in \Sigma^*$ and $n \geq N$, the relations $\xrightarrow{\mathbf{w}^n}$ and $\xrightarrow{\mathbf{w}^{n+1}}$ are the same.

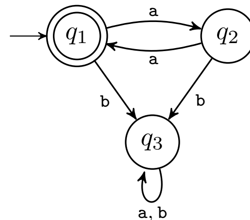
Example 1.8. Intuitively, a counter-free DFA is one that can test whether something happens, but not how many times it happens. For every cycle $q \xrightarrow{\mathbf{w}} q$, $\mathbf{w}$ cannot be written as $\mathbf{x}^k$ where $\mathbf{x} \in \Sigma^*$ and $k > 1$.

(a) The following DFA, which recognizes $(\texttt{ab})^*$, is counter-free:



This is counter-free because the only cycles are on ab (from q_1 to itself), a and b (from q_3 to itself), and none of these strings is of the form $\mathbf{x}^k$ for $k > 1$.

(b) The following DFA, which recognizes $(\texttt{aa})^*$, is not counter-free:



This is not counter-free because it has a cycle on $\texttt{aa}$, which is $\texttt{a}^2$.

Lemma 1.9. *If languages L_1 and L_2 are counter-free then*

1. *$L_1 \cup L_2$ is counter-free.*
2. *$L_1 - L_2$ is counter-free.*
3. *$L_1 \cap L_2$ is counter-free.*
4. *$L_1 \circ L_2$ is counter-free.*

Theorem 1.10 (Schützenberger, 1965, McNaughton and Papert, 1971)**.** *For any regular language L, the following are equivalent:*

- *L is star-free.*
- *L is counter-free.*
- *Its minimal DFA is counter-free.*

Thus, in Example 1.8(b), the DFA shown is the minimal DFA for $(\texttt{aa})^*$; since it is not counter-free, $(\texttt{aa})^*$ is not star-free.

First-order logic

A formal language can also be defined as a set of finite strings that satisfy a closed formula of a logic. This section is a significantly expanded version of Section 5.3 of the survey by Strobl et al. [2024b]; for more information, see the handbook chapter by Thomas [1997].

Example 1.11. Let $\Sigma = \{\texttt{a}, \texttt{b}\}$. The formula

$$\phi = \forall x.\forall y.Q_{\texttt{a}}(x) \wedge Q_{\texttt{b}}(y) \rightarrow x < y \tag{1.3}$$

defines the regular language $\texttt{a}^*\texttt{b}^*$. The variables (x, y) are interpreted as positions of a string $\mathbf{w}$, and $Q_{\texttt{a}}(i)$ is true iff $w_i = \texttt{a}$ and $Q_{\texttt{b}}(i)$ is true iff $w_i = \texttt{b}$. So the formula says that every $\texttt{a}$ must precede every $\texttt{b}$, which is true iff the string matches $\texttt{a}^*\texttt{b}^*$.

We first define the syntax of first-order logic with order (FO, sometimes written as $\mathsf{FO}[<]$ to emphasize the availability of an order relation $<$).

Definition 1.12. The *formulas* of FO are given by the BNF grammar:

$$\begin{aligned} \phi ::= {} & Q_\sigma(x) && \sigma \in \Sigma \\ & \mid x = y \mid x < y \\ & \mid \phi_1 \wedge \phi_2 \mid \phi_1 \vee \phi_2 \mid \neg\phi_1 \\ & \mid \forall x.\phi_1 \mid \exists x.\phi_1 \end{aligned} \tag{1.4}$$

where $x, y, \ldots$ are variables. The *free variables* of a formula are defined as follows:

$$\begin{aligned} \text{FV}(Q_\sigma(x)) &= \{x\} && \sigma \in \Sigma \\ \text{FV}(x = y) &= \{x, y\} \\ \text{FV}(x < y) &= \{x, y\} \\ \text{FV}(\phi_1 \wedge \phi_2) &= \text{FV}(\phi_1) \cup \text{FV}(\phi_2) \\ \text{FV}(\phi_1 \vee \phi_2) &= \text{FV}(\phi_1) \cup \text{FV}(\phi_2) \\ \text{FV}(\neg\phi_1) &= \text{FV}(\phi_1) \\ \text{FV}(\forall x.\phi_1) &= \text{FV}(\phi_1) \setminus \{x\} \\ \text{FV}(\exists x.\phi_1) &= \text{FV}(\phi_1) \setminus \{x\}. \end{aligned} \tag{1.5}$$

A formula is *closed* if it has no free variables. For example, $\mathrm{FV}(\exists y.x < y) = \{x\}$, while the formula in Eq. (1.3) is closed.

We use a number of shorthand notations:

$$\phi_1 \rightarrow \phi_2 = \neg\phi_1 \vee \phi_2 \qquad \text{implication} \tag{1.6}$$

$$\phi_1 \leftrightarrow \phi_2 = (\phi_1 \rightarrow \phi_2) \wedge (\phi_2 \rightarrow \phi_1) \qquad \text{if and only if} \tag{1.7}$$

$$\phi_1 \oplus \phi_2 = \neg(\phi_1 \leftrightarrow \phi_2) \qquad \text{exclusive or} \tag{1.8}$$

The semantics of FO defines whether formulas are true in various *logical structures*. Here, we are only interested in logical structures that correspond to strings. So we skip the definition of logical structure and go straight to the definition of whether a formula is true for a string.

Definition 1.13. Let $\mathbf{w} = w_0 \cdots w_{n-1}$ be a string over Σ, and let I be an *interpretation*, or a mapping from variables to $[n]$. We define $w, I \models \phi$ ("w and I satisfy ϕ") as follows:

$$\begin{aligned}
\mathbf{w}, I &\models Q_\sigma(x) && \text{if } w_{I(x)} = \sigma \\
\mathbf{w}, I &\models x = y && \text{if } I(x) = I(y) \\
\mathbf{w}, I &\models x < y && \text{if } I(x) < I(y) \\
\mathbf{w}, I &\models \phi_1 \wedge \phi_2 && \text{if } \mathbf{w}, I \models \phi_1 \text{ and } \mathbf{w}, I \models \phi_2[I] \\
\mathbf{w}, I &\models \phi_1 \vee \phi_2 && \text{if } \mathbf{w}, I \models \phi_1 \text{ or } \mathbf{w}, I \models \phi_2 \\
\mathbf{w}, I &\models \neg\phi_1 && \text{if } \mathbf{w}, I \not\models \phi_1 \\
\mathbf{w}, I &\models \forall x.\phi_1 && \text{if } \mathbf{w}, I[x \mapsto i] \models \phi_1 \text{ for all } i \in [n] \\
\mathbf{w}, I &\models \exists x.\phi_1 && \text{if } \mathbf{w}, I[x \mapsto i] \models \phi_1 \text{ for some } i \in [n]
\end{aligned} \tag{1.9}$$

Above, the notation $I[x \mapsto i]$ stands for the mapping that sends x to i, and sends any other variable y to $I(y)$.

If $\mathbf{w}, I \models \phi$, and ϕ is a closed formula, we can simply write $\mathbf{w} \models \phi$. The language defined by a closed formula ϕ is $L(\phi) = \{w \in \Sigma^* \mid \mathbf{w} \models \phi\}$.

Example 1.14. Let ϕ be as in Eq. (1.3):

$$\phi = \forall x.\forall y.Q_{\mathtt{a}}(x) \wedge Q_{\mathtt{b}}(y) \rightarrow x < y.$$

Here are some examples of strings and interpretations that do or don't satisfy ϕ.

$$\begin{aligned}
\mathtt{abb}, \{x \mapsto 0\} &\models Q_{\mathtt{a}}(x) \\
\mathtt{abb}, \{y \mapsto 0\} &\not\models Q_{\mathtt{b}}(y) \\
\mathtt{abb}, \{y \mapsto 1\} &\models Q_{\mathtt{b}}(y) \\
\mathtt{abb}, \{y \mapsto 2\} &\models Q_{\mathtt{b}}(y) \\
\mathtt{abb}, \{x \mapsto 0, y \mapsto 0\} &\not\models x < y \\
\mathtt{abb}, \{x \mapsto 0, y \mapsto 1\} &\models x < y \\
\mathtt{abb}, \{x \mapsto 0, y \mapsto 2\} &\models x < y \\
\mathtt{abb}, \{x \mapsto 0, y \mapsto 0\} &\models Q_{\mathtt{a}}(x) \wedge Q_{\mathtt{b}}(y) \rightarrow x < y \\
\mathtt{abb}, \{x \mapsto 0, y \mapsto 1\} &\models Q_{\mathtt{a}}(x) \wedge Q_{\mathtt{b}}(y) \rightarrow x < y \\
\mathtt{abb}, \{x \mapsto 0, y \mapsto 2\} &\models Q_{\mathtt{a}}(x) \wedge Q_{\mathtt{b}}(y) \rightarrow x < y \\
\mathtt{abb}, \{x \mapsto 0\} &\models \forall y.Q_{\mathtt{a}}(x) \wedge Q_{\mathtt{b}}(y) \rightarrow x < y
\end{aligned}$$

Example 1.15. As a slightly more complicated example of what can be defined in FO, let

$$\begin{aligned} \text{FIRST}(x) &= \neg\exists y.y < x && (1.10)\\ \text{SUCC}(x,y) &= x < y \wedge \neg\exists z.x < z < y && (1.11)\\ \text{LAST}(x) &= \neg\exists y.y > x && (1.12) \end{aligned}$$

Then

$$\begin{aligned} \phi = {} & (\forall x.\text{FIRST}(x) \rightarrow Q_{\mathtt{a}}(x)) && (1.13)\\ & \wedge (\forall x.\forall y.\text{SUCC}(x,y) \rightarrow ((Q_{\mathtt{a}}(x) \wedge Q_{\mathtt{b}}(y)) \vee (Q_{\mathtt{b}}(x) \wedge Q_{\mathtt{a}}(y))))\\ & \wedge (\forall y.\text{LAST}(x) \rightarrow Q_{\mathtt{b}}(x)) \end{aligned}$$

defines the language $(\mathtt{ab})^*$.

Lemma 1.16. *(The class of languages definable in)* FO *has the following closure properties:*

1. FO *is closed under concatenation: if* $L_1, L_2 \in \mathsf{FO}$*, then* $L_1 \circ L_2 \in \mathsf{FO}$.
2. FO *is closed under finite iteration: if* $L \in \mathsf{FO}$*, then* $L^i \in \mathsf{FO}$ *for all* $i \in \mathbb{N}$.

Theorem 1.17 (McNaughton and Papert, 1971)**.** FO *defines exactly the class of star-free regular languages.*

Example 1.18. As mentioned earlier, virtually all phonological phenomena are characterized by Star-Free languages, and therefore are describable with First-Order Logic. For example, one common phonological constraint is that every word must have exactly one primary stressed syllable. If we imagine our alphabet symbols are `p` for primary stress and `s` for secondary stress, we can describe this constraint simply:

$$\phi = (\exists x.Q_{\mathtt{p}}(x) \wedge \forall y.(Q_{\mathtt{p}}(y) \rightarrow x = y)) \tag{1.14}$$

Linear temporal logic

In *linear temporal logic* (LTL) [Kamp, 1968], each formula implicitly depends on a single time point (or position). Here, we use a variant that employs only the **since** operator, which is equally expressive as the complete version [Gabbay et al., 1980].

Example 1.19. Let $\Sigma = \{\mathtt{a}, \mathtt{b}, \mathtt{\#}\}$.

- The formula $\phi = Q_{\mathtt{\#}}$ defines the language $\Sigma^*\mathtt{\#}$, which contains all and only strings with a `#` in the *last* position.
- The formula $\phi = Q_{\mathtt{\#}} \wedge (Q_{\mathtt{b}}\ \textbf{since}\ Q_{\mathtt{\#}})$ defines the language $\Sigma^*\mathtt{\#}\mathtt{b}^*\mathtt{\#}$. You may read it as, "The last symbol is `#`, and since the previous `#`, it's been all `b`'s."
- The formula $\phi = Q_{\mathtt{\#}} \wedge (Q_{\mathtt{b}}\ \textbf{since}\ (Q_{\mathtt{\#}} \wedge (Q_{\mathtt{a}}\ \textbf{since}\ Q_{\mathtt{\#}})))$ defines the language $\Sigma^*\mathtt{\#}\mathtt{a}^*\mathtt{\#}\mathtt{b}^*\mathtt{\#}$.

Definition 1.20 (linear temporal logic)**.** The syntax of LTL is defined as follows:

$$\begin{aligned} \phi ::= {} & Q_\sigma && \sigma \in \Sigma\\ & \mid \phi_1 \wedge \phi_2 \mid \phi_1 \vee \phi_2 \mid \neg\phi_1 && (1.15)\\ & \mid \phi_1\ \textbf{since}\ \phi_2 \end{aligned}$$

For any input string $\mathbf{w} = w_0 \cdots w_{n-1}$ and position $i \in [n]$, we define $\mathbf{w}, i \models \phi$ as follows:

$$\begin{aligned}
&\mathbf{w}, i \models Q_\sigma && \text{if } w_i = \sigma \\
&\mathbf{w}, i \models \phi_1 \wedge \phi_2 && \text{if } \mathbf{w}, i \models \phi_1 \text{ and } \mathbf{w}, i \models \phi_2 \\
&\mathbf{w}, i \models \phi_1 \vee \phi_2 && \text{if either } \mathbf{w}, i \models \phi_1 \text{ or } \mathbf{w}, i \models \phi_2 \\
&\mathbf{w}, i \models \neg\phi_1 && \text{if } \mathbf{w}, i \not\models \phi_1 \\
&\mathbf{w}, i \models \phi_1 \textbf{ since } \phi_2 && \text{if there is a } j < i \text{ such that } \mathbf{w}, j \models \phi_2, \text{ and} \\
& && \quad \text{for all } k \text{ such that } j < k < i, \text{ we have } \mathbf{w}, k \models \phi_1
\end{aligned} \tag{1.16}$$

To use a formula ϕ of LTL to define a language over Σ, for a $\mathbf{w} \in \Sigma^*$ of length n we supply $\mathbf{w}$ as input and designate the last position as the output position, so that $\mathbf{w} \in \mathcal{L}(\phi)$ if and only if $\mathbf{w}, n-1 \models \phi$.

Perhaps surprisingly, LTL is exactly as expressive as FO over strings, and thus is another equivalent formalism to those we have seen so far.

Theorem 1.21 (Kamp 1968, Gabbay et al. 1980)**.** LTL *defines the exactly the class of star-free regular languages.*

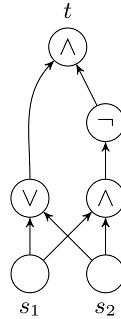
1.3.2 Circuit complexity

This section is an expanded version of Section 5.2 of the survey by Strobl et al. [2024b]. For more details, please see the textbook by Arora and Barak [2009].

For the rest of this chapter, we deal a lot with bits. We write $\log n$ for $\lceil \log_2 n \rceil$, which is the number of bits needed to represent a number in $[n]$.

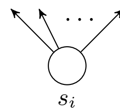
Circuits operate on binary values, so for the rest of this section, we assume $\Sigma = \{0, 1\}$. (If we want to use circuits to model sets of strings over an alphabet Σ, we can choose a fixed-length encoding of the symbols of Σ as strings of $b = \log |\Sigma|$ bits and encode the value of the i-th input symbol into positions ib to $ib + (b-1)$.)

Example 1.22. Here's a circuit with input length 2. It computes the XOR function. We draw the inputs at the bottom and the output at the top.

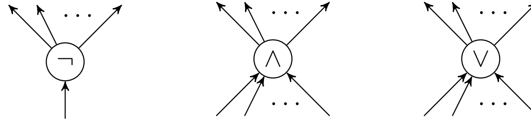


Definition 1.23 (Boolean circuits)**.** A *(Boolean) circuit C* with input length n is a directed acyclic procedural graph with

1. n nodes $s_0, \ldots, s_{n-1}$ with zero fan-in, designated as *input* nodes:

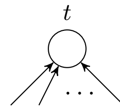


2. zero or more fan-in (in-degree) *gate* nodes, each labeled with a function:



- NOT ($\neg$), with fan-in one.
- AND ($\wedge$), with arbitrary fan-in. (An AND gate with fan-in zero always has value 1.)
- OR ($\vee$), with arbitrary fan-in. (An OR gate with fan-in zero always has value 0.)

3. A node t, which can be either an input or gate node, is designated the *output* of the circuit.

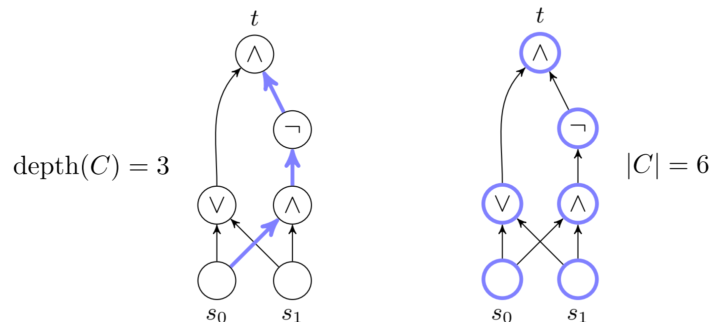


Given an input string $\mathbf{w} \in \{0,1\}^n$, each input node s_i is assigned the value w_i, and each gate node labeled f computes its value by applying f to the values of its in-neighbors.

We can think of the circuit as computing a Boolean function $C\colon \{0,1\}^n \to \{0,1\}$, mapping each input string to the value of t.

The *depth* of C, denoted $\mathrm{depth}(C)$, is the length of the longest directed path from any s_i to t. The *size* of C, denoted $|C|$, is the number of nodes in C.

Example 1.24. The longest path in C in Example 1.22 is 3, therefore our $\mathrm{depth}(C) = 3$. The number of nodes in C is 6, therefore $|C| = 6$.



Definition 1.25 (Boolean circuit families)**.** A *circuit family* is a sequence $\mathcal{C} = (C_n)_{n\in\mathbb{N}}$ such that for each n, C_n is a circuit with input length n. We treat $\mathcal{C}$ as a function on $\{0,1\}^*$ as follows.

For every $\mathbf{w} \in \{0,1\}^*$ with length n, $\mathcal{C}(\mathbf{w}) = C_n(\mathbf{w})$. Then the language defined by $\mathcal{C}$ is $L(\mathcal{C}) = \{\mathbf{w} \in \{0,1\}^* \mid \mathcal{C}(\mathbf{w}) = 1\}$. The *depth* and *size* of $\mathcal{C}$ are the functions $n \mapsto \mathrm{depth}(C_n)$ and $n \mapsto |C_n|$.

Since the depth and size of a circuit family are functions, we are interested in how they depend asymptotically on n. In particular, since transformers have constant depth, circuit classes with constant depth are of particular interest.

Definition 1.26 (AC^k and NC^k)**.** We define the following classes of languages:

- AC^k is the class of languages that can be recognized by families of circuits with unbounded fan-in, $O(\text{poly}(n))$ size, and $O((\log n)^k)$ depth.
- NC^k is the class of languages that can be recognized by families of circuits with fan-in at most 2, $O(\text{poly}(n))$ size, and $O((\log n)^k)$ depth.

A circuit family contains a different circuit for each length n, with no constraint on the relationship between the circuits. This has some surprising consequences.

Example 1.27. Let L be any *unary* language, that is, $L \subseteq \{1\}^*$. For each $n \in \mathbb{N}$, if $1^n \in L$, let C_n be a circuit that always has value 1 (an AND gate with fan-in zero), and if $1^n \notin L$, let C_n be a circuit that has value 0 (an OR gate with fan-in zero). Then, L is recognized by a circuit family with $O(n)$ size and $O(1)$ depth, and is therefore in AC^0, even if it is undecidable.

To prevent such consequences, we impose a $\mathsf{DLOGTIME}$-*uniform* restriction, which says that, given n, the circuit C_n can be constructed in logarithmic time, in the following sense.

Definition 1.28 (uniformity, Barrington et al., 1990)**.** Let $\mathcal{C} = (C_n)_{n\in\mathbb{N}}$ be a circuit family, and assume that the nodes of C_n are numbered from 0 to $|C_n| - 1$. We say that $\mathcal{C}$ is $\mathsf{DLOGTIME}$-*uniform* if there is a (deterministic) Turing machine that runs in logarithmic time and accepts those tuples $\langle f, i, j, \mathtt{1}^n\rangle$ such that in C_n, node i has label f and there is an edge from node i to node j.

We didn't include circuits in our tour of characterizations of star-free languages (Section 1.3.1), because star-free languages don't have a nice characterization in terms of circuits (to our knowledge), but a mild extension does:

Theorem 1.29 (Barrington et al., 1992)**.** $\mathsf{FO}[\text{MOD}]$*, which is* FO *extended with predicates* $\text{MOD}_{r,m}(x)$ *which are true just in case* $x \equiv r \pmod m$*, defines exactly the regular languages in* AC^0*.*

1.4 Transformers

In this section, we define transformers and relevant variants, and how transformers are used to describe formal languages. This section is adapted from Section 4 of the survey by Strobl et al. [2024b].

Transformers are composed of an input layer (Section 1.4.1), one or more hidden layers (Section 1.4.5), and an output layer. The inputs and outputs of the layers are sequences of vectors, which we treat as members of $(\mathbb{R}^d)^*$.

1.4.1 Input layer

Strings are initially mapped to sequences of vectors by $\mathit{emb}\colon \Sigma^* \xrightarrow{\text{lp}} (\mathbb{R}^d)^*$, which is the sum of a *word embedding* $\text{WE}\colon \Sigma \to \mathbb{R}^d$ and a *position(al) embedding* or *encoding* $\text{PE}_n\colon [n] \to \mathbb{R}^d$ for $n \in \mathbb{N}_{>0}$:

$$\mathit{emb}(w_0 \cdots w_{n-1})[i] = \text{WE}(w_i) + \text{PE}_n(i). \tag{1.17}$$

In theoretical constructions, the word embedding can be any computable function.

The original transformer paper [Vaswani et al., 2017] introduced the following position embedding:

$$\mathrm{PE}_n(i)[j] = \begin{cases} \sin(10000^{-j/d} \cdot i) & \text{if } j \text{ even} \\ \cos(10000^{-(j-1)/d} \cdot i) & \text{if } j \text{ odd.} \end{cases} \tag{1.18}$$

Theoretical papers have explored other position embeddings, including i itself [Pérez et al., 2021], i/n [Yao et al., 2021, Chiang and Cholak, 2022], and $1/i$ or $1/i^2$ [Pérez et al., 2021].

1.4.2 Attention

Scaled dot-product self-attention with d input/output dimensions and d_{hid} key/value dimensions is a function

$$\begin{aligned} att \colon (\mathbb{R}^d)^* &\overset{\text{lp}}{\to} (\mathbb{R}^d)^* \\ \mathbf{X} &\mapsto \mathbf{Y} && (1.19) \\ \mathbf{Q}[i] &= \mathbf{W}^{\mathrm{Q}}(\mathbf{X}[i]) && (1.20) \\ \mathbf{K}[j] &= \mathbf{W}^{\mathrm{K}}(\mathbf{X}[j]) && (1.21) \\ \mathbf{V}[j] &= \mathbf{W}^{\mathrm{V}}(\mathbf{X}[j]) && (1.22) \\ \mathbf{s}[i,j] &= \frac{\mathbf{Q}[i] \cdot \mathbf{K}[j]}{\sqrt{d_{\text{hid}}}} && (1.23) \\ \alpha[i,:] &= \mathrm{softmax}(\mathbf{s}[i,:]) && (1.24) \\ &= \frac{\exp \mathbf{s}[i,:]}{\sum_{j \in [d]} \exp \mathbf{s}[i,j]} && (1.25) \\ \mathbf{Y}[i] &= \sum_{j \in [n]} \alpha[i,j]\, \mathbf{V}[j] && (1.26) \end{aligned}$$

with parameters

$$\begin{aligned} \mathbf{W}^{\mathrm{Q}}, \mathbf{W}^{\mathrm{K}} &\in \mathbb{R}^{d_{\text{hid}} \times d} \\ \mathbf{W}^{\mathrm{V}} &\in \mathbb{R}^{d \times d} \end{aligned}$$

We call the $\mathbf{Q}[i]$ the *queries*, the $\mathbf{K}[j]$ the *keys*, the $\mathbf{V}[j]$ the *values*, the $\mathbf{s}[i,j]$ the *attention scores*, and we call the $\alpha[i,j]$ the *attention weights*.

Real transformers use *multi-head* self-attention, which is the sum of H attentions, $att(\mathbf{X}) = \sum_{i=1}^{H} att_i(\mathbf{X})$. This can be emulated as the composition of H many single-head attentions, each with different parameters, so we usually focus on single-head attention. However, when discussing the depth of transformers, the parallel structure of multi-head attention will become important.

Attention masking In *future-masked* (also known as *causally*-masked) self attention, a term $m(i,j)$ is added to Eq. (1.23) to force every position to attend only to preceding positions:

$$m(i,j) = \begin{cases} 0 & \text{if } j \le i \\ -\infty & \text{otherwise.} \end{cases} \tag{1.27}$$

(We define $\exp(-\infty) = 0$.) Some papers use *strict* future-masking, that is, $m(i,j) = 0$ iff $j < i$, and occasionally *past*-masking ($j \ge i$) and strict past-masking ($j > i$).

1.4.3 Feed-forward networks

Although feed-forward networks (FFNNs) come in many varieties, for our purposes they always look like this:

Definition 1.30 (linear layer)**.** A *linear layer* is a function

$$\begin{aligned} lin\colon \mathbb{R}^d &\to \mathbb{R}^{d'} \\ \mathbf{x} &\mapsto \mathbf{W}\mathbf{x} + \mathbf{b} \end{aligned} \tag{1.28}$$

with attributes

- $d \in \mathbb{N}$, called the *input size*
- $d' \in \mathbb{N}$, called the *output size*

and parameters

- $\mathbf{W} \in \mathbb{R}^{d' \times d}$, called the *weights*
- $\mathbf{b} \in \mathbb{R}^{d'}$, called the *bias*.

Definition 1.31 (feed-forward neural network)**.** A *feed-forward neural network* (FFNN) is a function

$$\begin{aligned} \mathit{ffn}\colon \mathbb{R}^{d_{\text{hid}}} &\to \mathbb{R}^{d_{\text{hid}}} \\ \mathbf{x} &\mapsto \mathbf{y} \text{ where} \\ \mathbf{h} &= \text{ReLU}(lin_1(\mathbf{x})) \\ \mathbf{y} &= lin_2(\mathbf{h}) \end{aligned} \tag{1.29}$$

with attributes

- $d_{\text{FFN}} \in \mathbb{N}$, commonly set to $4d_{\text{hid}}$, but theoretical constructions use as many or as few dimensions as needed.
- linear layers lin_1 and lin_2 such that

$$\begin{aligned} lin_1.d = lin_2.d' = d_{\text{hid}} \\ lin_1.d' = lin_2.d = d_{\text{FFN}}. \end{aligned}$$

(Recall the dot notation introduced on page 3.)

The following expressivity results will come in handy. They are special cases of a theorem that says that any continuous piecewise linear function on $\mathbb{R}^d$ can be computed by a FFNN with $O(\log d)$ layers [Arora et al., 2018]. But we are restricting FFNNs to have two layers.

Theorem 1.32. *Any Boolean function can be computed by a FFNN.*

Proof. Any Boolean function f can be converted to *full disjunctive normal form (DNF)*, in which at most one of the clauses can be true. Although it's not necessarily the most efficient, we can always do this by listing out all the inputs that make f true:

$$f(x_0, \ldots, x_{n-1}) = \bigvee_{\substack{\xi_0, \ldots, \xi_{n-1} \in \{0,1\} \\ f(\xi_0, \ldots, \xi_{n-1}) = 1}} (x_0 \leftrightarrow \xi_0 \wedge \cdots \wedge x_{n-1} \leftrightarrow \xi_{n-1}).$$

For example, the XOR function looks like

$$\begin{aligned}\text{XOR}(x,y) &= (x \leftrightarrow 0 \wedge y \leftrightarrow 1) \vee (x \leftrightarrow 1 \wedge y \leftrightarrow 0) \\ &= (\neg x \wedge y) \vee (x \wedge \neg y).\end{aligned}$$

Then we construct a 2-layer FFNN *ffn* with ReLU activations. It has 2^n hidden units:

$$h_{\xi_0,\ldots,\xi_{n-1}} = \text{ReLU}\left(\sum_k \lambda_{\xi_k,k} - n + 1\right) \qquad \xi_0,\ldots,\xi_{n-1} \in \{0,1\} \tag{1.30}$$

$$\lambda_{\xi,k} = \begin{cases} x_k & \xi = 1 \\ 1 - x_k & \xi = 0 \end{cases} \qquad k \in [n] \tag{1.31}$$

$$y = \sum_{\substack{\xi_0,\ldots,\xi_{n-1}\in\{0,1\} \\ f(\xi_0,\ldots,\xi_{n-1})=1}} h_{\xi_0,\ldots,\xi_{n-1}}. \tag{1.32}$$

It's because we know that at most one clause is true that we can compute the disjunction using a simple addition (Eq. (1.32)). □

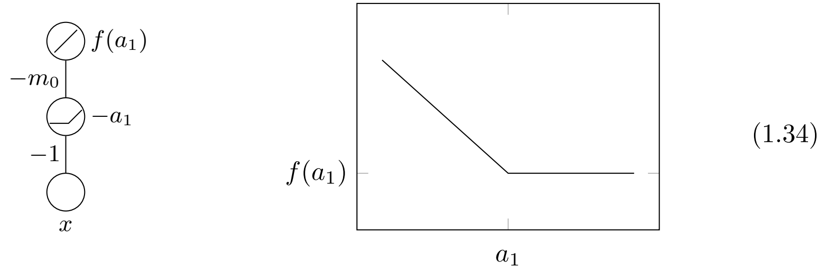
Theorem 1.33. *On scalar (that is, size 1) inputs, FFNNs compute exactly the set of continuous piecewise linear functions with a finite number of pieces (or CPWL functions for short).*

Proof. Assume that f has the form:

$$f(x) = \begin{cases} m_0 x + b_0 & x < a_1 \\ m_1 x + b_1 & a_1 \le x < a_2 \\ \vdots & \\ m_{k-1} x + b_{k-1} & a_{k-1} \le x < a_k \\ m_k x + b_k & a_k \le x. \end{cases} \tag{1.33}$$

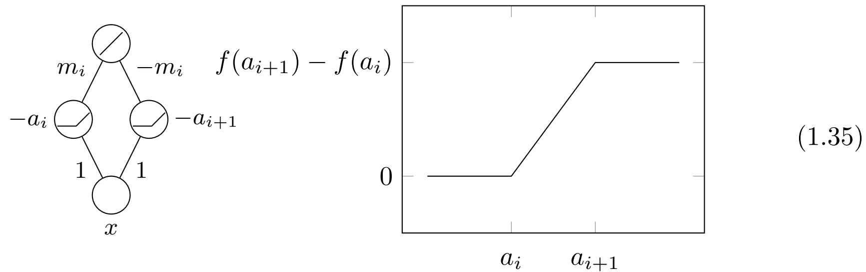
where $m_{i-1}a_i + b_{i-1} = m_i a_i + b_i$ for all i.

The first piece is made of a ReLU flipped left-to-right:



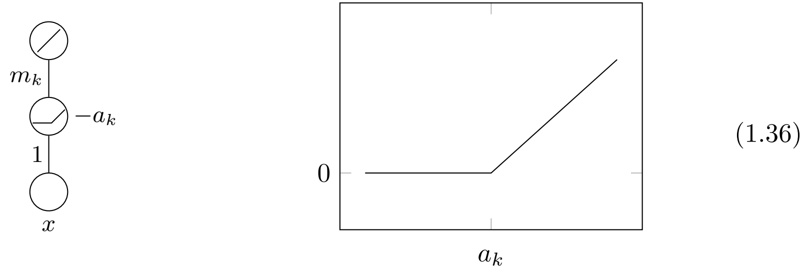
(1.34)

For each middle piece ($i = 1,\ldots,k-1$), we add in a "wedge" (also known as a *saturated linear unit*) like this:



(1.35)

Finally, we add in one more ReLU for the last piece:



(1.36)

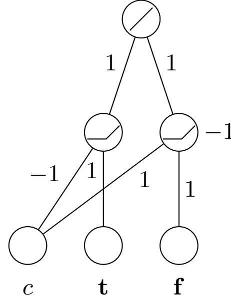
□

Theorem 1.34. *For any $d > 0$ there is a FFNN that computes*

$$\begin{aligned} \text{if} \colon \mathbb{R}^{2d+1} &\to \mathbb{R}^d \\ \begin{bmatrix} c \\ \mathbf{t} \\ \mathbf{f} \end{bmatrix} &\mapsto \begin{cases} \mathbf{t} & \text{if } c = 0 \\ \mathbf{f} & \text{if } c = 1 \end{cases} \end{aligned}$$

provided $0 \leq \mathbf{t}[i] \leq 1$ and $0 \leq \mathbf{f}[i] \leq 1$ for all i.

Proof. The FFNN is [Pérez et al., 2021, Merrill and Sabharwal, 2024]:



□

1.4.4 Layer normalization

A d-dimensional *layer normalization* [Ba et al., 2016], or *layernorm* for short, is a function

$$\begin{aligned} \textit{norm} \colon \mathbb{R}^d &\to \mathbb{R}^d \\ \mathbf{x} &\mapsto \gamma \odot \frac{\mathbf{x} - \bar{\mathbf{x}}}{\sqrt{\mathrm{Var}(\mathbf{x}) + \varepsilon}} + \beta \end{aligned} \tag{1.37}$$

where $\odot$ is component-wise multiplication,

$$\bar{\mathbf{x}} = \frac{1}{d} \sum_{i \in [d]} \mathbf{x}_i \tag{1.38}$$

$$\mathrm{Var}(\mathbf{x}) = \frac{1}{d} \sum_{i \in [d]} (\mathbf{x}_i - \bar{\mathbf{x}})^2 \tag{1.39}$$

and the parameters are

$$\begin{aligned} \gamma, \beta &\in \mathbb{R}^d \\ \varepsilon &\geq 0. \end{aligned}$$

The original definition of layernorm [Ba et al., 2016] sets $\varepsilon = 0$, but, for numerical stability, and to avoid division by zero, all implementations we are aware of set $\varepsilon > 0$. Observe that *norm* is Lipschitz-continuous iff $\varepsilon > 0$.

Some transformer analyses omit layernorm for simplicity [e.g. Pérez et al., 2021].

1.4.5 Hidden layers

A *transformer layer* (also known as a *block*) comes in two variants. The *post-norm* variant [Vaswani et al., 2017] is

$$\begin{aligned} \mathit{layer} \colon (\mathbb{R}^d)^* &\xrightarrow{\text{lp}} (\mathbb{R}^d)^* \\ \mathbf{X} &\mapsto \mathbf{Y} \text{ where} \\ \mathbf{H} &= \mathit{norm}_1(\mathbf{X} + \mathit{att}(\mathbf{X})) \\ \mathbf{Y} &= \mathit{norm}_2(\mathbf{H} + \mathit{ffn}(\mathbf{H})) \end{aligned} \tag{1.40}$$

and the *pre-norm* variant [Wang et al., 2019] has

$$\begin{aligned} \mathbf{H} &= \mathbf{X} + \mathit{att}(\mathit{norm}_1(\mathbf{X})) \\ \mathbf{Y} &= \mathbf{H} + \mathit{ffn}(\mathit{norm}_2(\mathbf{H})) \end{aligned} \tag{1.41}$$

where

- *att* is a self-attention with d input/output dimensions and d_{hid} key/value dimensions
- *ffn* is a feed-forward network with d input/output dimensions and two layers, one ReLU and one linear.
- norm_1 and norm_2 are layernorms with d dimensions.

In both variants, the $\mathbf{X} +$ and $\mathbf{H} +$ terms are called *residual connections* [He et al., 2015], also known as *skip connections*.

1.4.6 Transformer encoders

A *post-norm transformer encoder* is a length-preserving function

$$\begin{aligned} \mathit{tfr} \colon \Sigma^* &\xrightarrow{\text{lp}} (\mathbb{R}^d)^* \\ \mathbf{w} &\mapsto \mathbf{A}^{(L)} \text{ where} \\ \mathbf{A}^{(0)} &= \mathit{emb}(\mathbf{w}) \\ \mathbf{A}^{(1)} &= \mathit{layer}_1(\mathbf{A}^{(0)}) \\ &\vdots \\ \mathbf{A}^{(L)} &= \mathit{layer}_L(\mathbf{A}^{(L-1)}) \end{aligned} \tag{1.42}$$

where

- *emb* is an input layer
- L is called the *depth*
- each layer_ℓ for $\ell \in 1, \dots, L$ is a post-norm transformer layer (1.40).

A *pre-norm* transformer encoder is additionally parameterized by the weights of a final layernorm *norm* and is defined as:

$$\begin{aligned} \mathit{tfr} \colon \Sigma^* &\xrightarrow{\text{lp}} (\mathbb{R}^d)^* \\ \mathbf{w} &\mapsto \mathit{norm}(\mathbf{A}^{(L)}) \text{ where} \\ \mathbf{A}^{(0)} &= \mathit{emb}(\mathbf{w}) \\ \mathbf{A}^{(1)} &= \mathit{layer}_1(\mathbf{A}^{(0)}) \\ &\vdots \\ \mathbf{A}^{(L)} &= \mathit{layer}_L(\mathbf{A}^{(L-1)}) \end{aligned} \tag{1.43}$$

where

- *emb* is an input layer
- L is called the *depth*
- each layer_ℓ for $\ell \in 1, \dots, L$ is a pre-norm transformer layer (1.41)
- *norm* is a layernorm.

The encoder's output is a sequence of vectors in $(\mathbb{R}^d)^*$. To use it as a language recognizer, we add a linear output layer

$$\begin{aligned} \mathit{out} \colon \mathbb{R}^d &\to \mathbb{R} \\ \mathbf{h} &\mapsto \mathbf{W}\mathbf{h} + b \qquad\qquad i \in [n] \end{aligned} \tag{1.44}$$

with parameters $\mathbf{W} \in \mathbb{R}^{1 \times d}$ and $b \in \mathbb{R}$. The encoder accepts iff $\mathit{out}(\mathit{tfr}(\mathbf{w})[n]) \geq 0$.

1.4.7 Remarks

Encoders, decoders, and encoder–decoders. Besides transformer encoders, there are other varieties. Transformer *decoders*, which will be introduced in Section 3.1.2, generate strings instead of recognizing them. The original variety [Vaswani et al., 2017] was a transformer encoder–decoder for machine translation: the encoder reads in a source-language string and the decoder generates a target-language string.

Transformers and circuits. In Section 1.3 we mentioned that out of the various formalisms we introduced, circuits have the most transparent connection with neural networks. A neural network is essentially an arithmetic circuit with some fancy operations (also known as a computation graph): a graph in which each node is an operation $\mathbb{R}^k \to \mathbb{R}$ (addition, multiplication, division, ReLU, exp). The concept of the depth of a neural network corresponds closely with the concept of depth of a circuit. As transformers have bounded depth, we expect that their circuit-family counterparts would also have bounded depth. The class NC^0 is not interesting (exercise: why?) but AC^0 will be our first circuit complexity class of interest. Later, we will introduce another circuit complexity class, TC^0, which is even more relevant.

Chapter 2

Encoders with Unique-Hard Attention

In this chapter, we consider transformers (transformer encoders) that use *unique-hard attention* (UHATs) instead of the standard *softmax attention*. Intuitively, unique-hard attention always concentrates all attention on a single position. Although this is simplistic compared to the way that transformers actually work, we will see that this assumption makes it possible to exactly characterize what languages unique-hard attention transformers can recognize, and to prove various facts about them; for example, that it is always possible to increase their expressivity by increasing their depth.

2.1 Unique-Hard Attention

For any vector $\mathbf{x} \in \mathbb{R}^d$, define $M(\mathbf{x}) = \{i \in [n] \mid \forall j \in [n], \mathbf{x}[j] \leq \mathbf{x}[i]\}$ to be the set of indices of the maximal elements of $\mathbf{x}$. In *leftmost-hard* attention, the leftmost maximal element is used, replacing Eq. (1.24) with:

$$\alpha[i,j] = \mathbb{I}[j = \min M(\mathbf{s}[i,:])] \tag{2.1}$$

whereas in *rightmost-hard* attention, the rightmost maximal element is used:

$$\alpha[i,j] = \mathbb{I}[j = \max M(\mathbf{s}[i,:])]. \tag{2.2}$$

Leftmost-hard attention was previously called *hard* attention by Hahn [2020] and *unique-hard* attention by Hao et al. [2022]. Here, we use the term *unique-hard attention* to refer to either leftmost-hard or rightmost-hard attention.

2.2 Overview

Unique-hard attention has been studied in several papers:

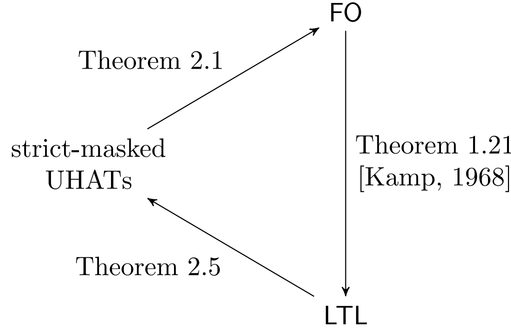
- Hahn [2020] introduced UHATs and proved that they cannot recognize PARITY.
- Hao et al. [2022] generalized the above result by showing that UHATs only recognize languages in AC^0.
- Barceló et al. [2024] proved that UHATs can recognize all languages in FO.

Here, we look at the results of Yang et al. [2023], which exactly characterize transformers with rightmost-hard attention and *strict* future masking. We will call these masked hard-attention transformers, or MUHATs for short.

Strict future masking means that each position i attends to positions $j < i$. If i is the leftmost position, all positions are masked out, so (following Merrill and Sabharwal [2024]) the attention output is just the zero vector.

2.3 Main Result

We're going to prove the equivalence differently from Yang et al. [2023]:



Theorem 2.1. *For any transformer encoder T with rightmost hard attention and strict future masking, there is a closed formula of* FO *that defines the same language that T recognizes.*

The proof hinges on the fact that in a unique hard attention transformer, each activation vector depends on at most two vectors from the layer below. Because the network has fixed, finite depth, there is a fixed, finite number of possible activation vectors that it can compute.

Lemma 2.2. *Let T be a unique (leftmost or rightmost) hard attention transformer. There is a finite set $\mathbb{F} \subseteq \mathbb{R}^d$ such that for any input string $\mathbf{w}$, all the activation vectors computed by $T(\mathbf{w})$ belong to $\mathbb{F}$.*

Proof. We prove that the self-attention at layer ℓ has at most $(|\Sigma|+1)^{2^\ell} - 1$ different possible output vectors, by induction on ℓ.

Base case ($\ell = 0$): Since there are no position embeddings, the embedding at position i is determined entirely by w_i, so there are at most $|\Sigma|$ possible activation vectors.

Inductive step ($\ell > 0$): Assume that the output of the layer $(\ell - 1)$ has at most $(|\Sigma|+1)^{2^{\ell-1}} - 1$ possible activation vectors, and consider layer ℓ:

- The attention output at position i depends only on $\mathbf{A}^{(\ell)}[i]$ (because of the residual connection) and $\mathbf{A}^{(\ell)}[j_i]$ (where j_i is the position that i attends to). So the number of possible output activation vectors is at most

$$\begin{aligned}\left((|\Sigma|+1)^{2^{\ell-1}} - 1\right)(|\Sigma|+1)^{2^{\ell-1}} &\leq \left((|\Sigma|+1)^{2^{\ell-1}} - 1\right)\left((|\Sigma|+1)^{2^{\ell-1}} + 1\right) \\ &= \left((|\Sigma|+1)^{2^{\ell-1}}\right)^2 - 1 \\ &= (|\Sigma|+1)^{2^\ell} - 1.\end{aligned}$$

- Because the FFNN and layernorms operate position-wise, they also have at most $(|\Sigma|+1)^{2^\ell}-1$ possible activation vectors.

Therefore, the number of possible output vectors from a self-attention or FFNN at layer ℓ is at most $(|\Sigma|+1)^{2^\ell}-1$.

Then $\mathbb{F}$ is the union over all layers of the possible activation vectors. □

Although it's not the most efficient way to do it, we can represent each value computed by the network as a set of $|\mathbb{F}|$ many formulas. (Yang et al. [2023] show how to do this with only $O(\log|\mathbb{F}|)$ many formulas.) Then any positionwise function can be defined by writing a formula that is essentially a lookup table.

Definition 2.3. Let $\mathbb{F}$ be any finite set. A function $\mathbf{X}\colon \Sigma^* \xrightarrow{\text{lp}} \mathbb{F}^*$ is definable by FO formulas $(\phi_{\mathbf{X}=\mathbf{v}}(i))_{\mathbf{v}\in\mathbb{F}}$ if for all $\mathbf{w}\in\Sigma^*$ and $\mathbf{v}\in\mathbb{F}$, we have $\mathbf{X}(\mathbf{w})[i]=\mathbf{v}$ iff $\mathbf{w}\models\phi_{\mathbf{X}=\mathbf{v}}(i)$.

Lemma 2.4. *Let $\mathbb{F}$ and $\mathbb{F}'$ be any finite sets. If $\mathbf{X}\colon \Sigma^* \xrightarrow{\text{lp}} \mathbb{F}^*$ is definable by FO formulas $(\phi_{\mathbf{X}=\mathbf{v}}(i))_{\mathbf{v}\in\mathbb{F}}$, and $f\colon\mathbb{F}\to\mathbb{F}'$, then there are FO formulas $(\phi_{f(\mathbf{X})=\mathbf{v}}(i)))_{\mathbf{v}\in\mathbb{F}}$ that define $\mathbf{w}\mapsto f(\mathbf{X}(\mathbf{w}))$, where f is applied positionwise.*

Proof. For all $\mathbf{v}\in\mathbb{F}'$, define

$$\phi_{f(\mathbf{X})=\mathbf{v}}(i) = \bigvee_{\substack{\mathbf{u}\in\mathbb{F}\\ f(\mathbf{u})=\mathbf{v}}} \phi_{\mathbf{X}=\mathbf{u}}(i). \tag{2.3}$$

□

Proof of Theorem 2.1. For every activation value $\mathbf{A}^{(\ell)}[i]$ (which depends on $\mathbf{w}$ and can be thought of as a function of $\mathbf{w}$), we will construct a formula $\mathrm{act}_{\ell,\mathbf{v}}(i)$ such that $\mathbf{w}\models\mathrm{act}_{\ell,\mathbf{v}}(i)$ iff $\mathbf{A}^{(\ell)}[i]=\mathbf{v}$. We do this by induction on ℓ.

Case $\ell=0$: At the bottom of the network, we have the embedding layer, $\mathbf{A}^{(0)}[i]=\mathrm{WE}(\mathbf{w}[i])$. We define this in FO as

$$\mathrm{act}_{0,\mathbf{v}}(i) = \bigvee_{\substack{a\in\Sigma\\ \mathrm{WE}(a)=\mathbf{v}}} Q_a(i). \tag{2.4}$$

Case $\ell>0$: Assume that there are formulas $\mathrm{act}_{\ell-1,\mathbf{v}}(i)$ that define the first ℓ layers, ending in activations $\mathbf{A}^{(\ell-1)}[i]$ (Eq. (1.42) or Eq. (1.43)). Our goal is to write formulas that define $\mathbf{A}^{(\ell)}[i]$. We first need to translate the self-attention into FO, starting with Eq. (1.23) with $\mathbf{X}=\mathbf{A}^{(\ell-1)}$.

By Lemma 2.4, there are formulas $\mathrm{query}_{\ell,\mathbf{v}}(i)$, $\mathrm{key}_{\ell,\mathbf{v}}(i)$, and $\mathrm{value}_{\ell,\mathbf{v}}(i)$ such that

$$\mathbf{w}\models\mathrm{query}_{\ell,\mathbf{v}}(i) \quad\text{iff}\quad \mathbf{W}^{\mathrm{Q}}(\mathbf{A}^{(\ell-1)})[i]=\mathbf{v} \tag{2.5}$$

$$\mathbf{w}\models\mathrm{key}_{\ell,\mathbf{v}}(i) \quad\text{iff}\quad \mathbf{W}^{\mathrm{K}}(\mathbf{A}^{(\ell-1)})[i]=\mathbf{v} \tag{2.6}$$

$$\mathbf{w}\models\mathrm{value}_{\ell,\mathbf{v}}(i) \quad\text{iff}\quad \mathbf{W}^{\mathrm{V}}(\mathbf{A}^{(\ell-1)})[i]=\mathbf{v}. \tag{2.7}$$

By analogy with Lemma 2.4, there are formulas $\mathrm{score}_{\ell,v}(i,j)$ that define $\mathbf{s}^{(\ell)}[i,j]$, the attention score at layer ℓ from position i to position j:

$$\mathrm{score}_{\ell,s}(i,j) = \bigvee_{\substack{\mathbf{u},\mathbf{v}\in\mathbb{F}\\ \mathbf{u}\cdot\mathbf{v}=s}} \left(\mathrm{query}_{\ell,\mathbf{u}}(i)\wedge\mathrm{key}_{\ell,\mathbf{v}}(j)\right) \tag{2.8}$$

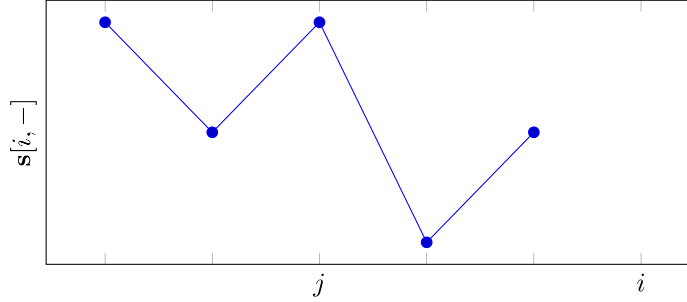
where s ranges over possible scores (a finite set). Furthermore, there is a formula $\text{compare}_\ell(i, j_1, j_2)$ that holds iff $\mathbf{s}^{(\ell)}[i, j_1] \leq \mathbf{s}^{(\ell)}[i, j_2]$:

$$\text{compare}_\ell(i, j_1, j_2) = \bigvee_{\substack{s_1, s_2 \\ s_1 \leq s_2}} (\text{score}_{\ell, s_1}(i, j_1) \wedge \text{score}_{\ell, s_2}(i, j_2)). \tag{2.9}$$

Then we can define a formula that tests whether position i attends to position j (that is, $\text{weight}_\ell(i, j)$ holds iff $\alpha[i, j]$ at layer ℓ).

$$\begin{aligned} \text{weight}_\ell(i, j) = j < i & \\ & \wedge \forall k[k < j \rightarrow \text{compare}_\ell(i, k, j)] \\ & \wedge \forall k[j < k \wedge k < i \rightarrow \neg\text{compare}_\ell(i, j, k)]. \end{aligned} \tag{2.10}$$

Because j is the rightmost position with maximal score, the positions left of j must have score less than or equal to $\mathbf{s}[i, j]$, but the positions between j and i (exclusive) must have score strictly less than $\mathbf{s}[i, j]$. For example:



Next we can write formulas that define the attention output at position i (which is $\mathbf{Y}[i]$ in Eq. (1.26)), taking care to deal with the edge case $i = 0$:

$$\text{att}_{\ell, \mathbf{v}}(i) = \begin{cases} \exists j[\text{weight}_\ell(i, j) \wedge \text{value}_{\ell, \mathbf{v}}(j)] & \mathbf{v} \neq \mathbf{0} \\ \exists j[\text{weight}_\ell(i, j) \wedge \text{value}_{\ell, \mathbf{v}}(j)] \vee \neg\exists j[j < i] & \mathbf{v} = \mathbf{0}. \end{cases} \tag{2.11}$$

By Lemma 2.4 again, there are formulas $\text{act}_{\ell, \mathbf{v}}(i)$ that encode the residual connections and the FFNN (Eq. (1.40) or Eq. (1.41)), defining $\mathbf{A}^{(\ell)}[i]$. This completes the induction.

At the top of the network, we use Lemma 2.4 one last time to write formulas $\text{out}_v(i)$ that define the output layer, and finally we write a formula that tests the output at the last position:

$$\phi = \exists n.(\forall i.i \leq n) \wedge \bigvee_{v \geq 0} \text{out}_v(n).$$

□

To go in the other direction, we use the fact that LTL is equivalent to FO [Kamp, 1968] and do an easier conversion from LTL.

Theorem 2.5. *For any formula ϕ of* LTL*, there is a transformer encoder with rightmost hard attention and strict future masking that recognizes the same language that ϕ defines.*

The proof will be by induction on the structure of ϕ, so we need the following lemma for combining the translations of sister subformulas.

Lemma 2.6 (Parallel composition). *Given transformer encoders without layer normalization*

$$tfr_1 \colon \Sigma^* \xrightarrow{\text{lp}} (\mathbb{R}^{d_1})^*$$
$$tfr_2 \colon \Sigma^* \xrightarrow{\text{lp}} (\mathbb{R}^{d_2})^*$$

there is a transformer

$$tfr_1 \oplus tfr_2 \colon \Sigma^* \xrightarrow{\text{lp}} (\mathbb{R}^{d_1+d_2})^*$$

such that for all strings $\mathbf{w} = w_0 \cdots w_{n-1} \in \Sigma^*$,

$$(tfr_1 \oplus tfr_2)(\mathbf{w}) = \begin{bmatrix} tfr_1(\mathbf{w})[0] \\ tfr_2(\mathbf{w})[0] \end{bmatrix} \cdots \begin{bmatrix} tfr_1(\mathbf{w})[n-1] \\ tfr_2(\mathbf{w})[n-1] \end{bmatrix}. \tag{2.12}$$

Proof. Let d_1 and d_2 be the width of tfr_1 and tfr_2, and let $d = d_1 + d_2$. If one of tfr_1 and tfr_2 has fewer layers than the other, add trivial layers (layers that compute the identity function) until they have the same number of layers L.

The new transformer has embedding layer

$$(tfr_1 \oplus tfr_2).emb(\mathbf{w}) = \begin{bmatrix} tfr_1.emb(\mathbf{w})[0] \\ tfr_2.emb(\mathbf{w})[0] \end{bmatrix} \cdots \begin{bmatrix} tfr_1.emb(\mathbf{w})[n-1] \\ tfr_2.emb(\mathbf{w})[n-1] \end{bmatrix}.$$

For each layer $\ell \in [L]$, let $f_1 = tfr_1.layer_\ell$ and $f_2 = tfr_2.layer_\ell$. Widen f_1 into a layer f_1' with width d as follows.

$$f_1'.\mathbf{W}^{\mathrm{Q}} = \begin{bmatrix} f_1.\mathbf{W}^{\mathrm{Q}} & \mathbf{0} \end{bmatrix} \qquad f_1'.\mathbf{W}^{\mathrm{K}} = \begin{bmatrix} f_1.\mathbf{W}^{\mathrm{K}} & \mathbf{0} \end{bmatrix}$$
$$f_1'.\mathbf{W}^{\mathrm{V}} = \begin{bmatrix} f_1.\mathbf{W}^{\mathrm{V}} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} \end{bmatrix}$$
$$f_1'.lin_1.\mathbf{W} = \begin{bmatrix} f_1.lin_1.\mathbf{W} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} \end{bmatrix} \qquad f_1'.lin_1.\mathbf{b} = \begin{bmatrix} f_1.lin_1.\mathbf{b} \\ \mathbf{0} \end{bmatrix}$$
$$f_1'.lin_2.\mathbf{W} = \begin{bmatrix} f_1.lin_2.\mathbf{W} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} \end{bmatrix} \qquad f_1'.lin_2.\mathbf{b} = \begin{bmatrix} f_1.lin_2.\mathbf{b} \\ \mathbf{0} \end{bmatrix}$$

Similarly, widen f_2 into a layer f_2' with width d, but using the bottom half of the activation vectors:

$$f_2'.\mathbf{W}^{\mathrm{Q}} = \begin{bmatrix} \mathbf{0} & f_2.\mathbf{W}^{\mathrm{Q}} \end{bmatrix} \qquad f_2'.\mathbf{W}^{\mathrm{K}} = \begin{bmatrix} \mathbf{0} & f_2.\mathbf{W}^{\mathrm{K}} \end{bmatrix}$$
$$f_2'.\mathbf{W}^{\mathrm{V}} = \begin{bmatrix} \mathbf{0} & \mathbf{0} \\ \mathbf{0} & f_2.\mathbf{W}^{\mathrm{V}} \end{bmatrix}$$
$$f_2'.lin_1.\mathbf{W} = \begin{bmatrix} \mathbf{0} & \mathbf{0} \\ \mathbf{0} & f_2.lin_1.\mathbf{W} \end{bmatrix} \qquad f_2'.lin_1.\mathbf{b} = \begin{bmatrix} \mathbf{0} \\ f_2.lin_1.\mathbf{b} \end{bmatrix}$$
$$f_2'.lin_2.\mathbf{W} = \begin{bmatrix} \mathbf{0} & \mathbf{0} \\ \mathbf{0} & f_2.lin_2.\mathbf{W} \end{bmatrix} \qquad f_2'.lin_2.\mathbf{b} = \begin{bmatrix} \mathbf{0} \\ f_2.lin_2.\mathbf{b} \end{bmatrix}$$

Then stack f_1' on top of f_2', or the other way around. (If we had multi-head attention, we could have combined f_1 and f_2 into a single layer.) □

Proof of Theorem 2.5. For any LTL formula ϕ, there is a transformer tfr_ϕ and an index k such that $tfr_\phi(\mathbf{w})[i,k] = \mathbb{I}[\mathbf{w}, i \models \phi]$. We show this by induction on the structure of ϕ.

Base case $\phi = Q_a$ for some $a \in \Sigma$: Then tfr_ϕ is just an embedding function

$$\mathit{tfr}_\phi(\mathbf{w})[i] = \left[\mathbb{I}[w_i = a]\right]. \tag{2.13}$$

Case $\phi = \neg\phi_1$: By the induction hypothesis, there is a transformer tfr_1 simulating ϕ_1. For simplicity, we write the output activation vector of tfr_{ϕ_1} as

$$\mathit{tfr}_{\phi_1}(\mathbf{w})[i] = \begin{bmatrix} \mathbb{I}[\mathbf{w}, i \models \phi_1] \\ 0 \end{bmatrix} \tag{2.14}$$

even though $\mathit{tfr}_{\phi_1}(\mathbf{w})[i]$ presumably has other components not shown. We add a trivial self-attention layer and, by Theorem 1.32, a FFNN to simulate $\phi = \neg\phi_1$:

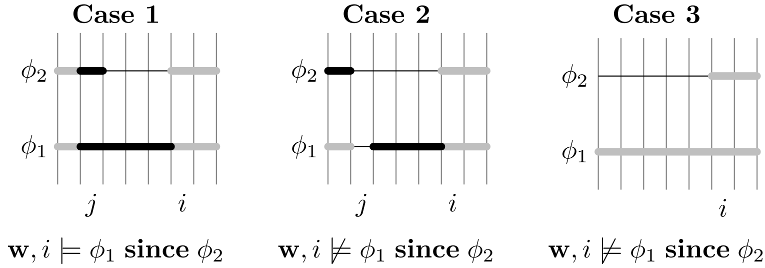
$$\mathit{ffn}_\neg(\mathit{tfr}_{\phi_1}(\mathbf{w}))[i] = \begin{bmatrix} 0 \\ \mathbb{I}[\mathbf{w}, i \models \neg\phi_1] \end{bmatrix}. \tag{2.15}$$

Case $\phi = \phi_1 \wedge \phi_2$, $\phi_1 \vee \phi_2$: By the induction hypothesis, there are transformers tfr_1 and tfr_2 simulating ϕ_1 and ϕ_2, respectively. Combine these using Lemma 2.6, add a trivial self-attention layer and, by Theorem 1.32, a FFNN to simulate ϕ.

Case $\phi = \phi_1$ **since** ϕ_2: By the induction hypothesis, there are transformers tfr_1 and tfr_2 simulating ϕ_1 and ϕ_2, respectively. Combine these using Lemma 2.6. For simplicity, we write the output activation vectors of the composed transformer as:

$$\mathbf{A}[i] = \begin{bmatrix} \mathbb{I}[\mathbf{w}, i \models \phi_1] \\ \mathbb{I}[\mathbf{w}, i \models \phi_2] \\ 0 \\ 0 \end{bmatrix}. \tag{2.16}$$

Imagine, given position i, how you would decide whether ϕ_1 **since** ϕ_2 is true. You could look at positions $i-1, i-2$, and so on. If ϕ_1 is true and ϕ_2 is false, then keep looking to the left. But when you encounter one of these situations, you know whether ϕ_1 **since** ϕ_2 is true (here, black means true, blank means false, and gray means either):



In case 1, the rightmost j satisfying ϕ_2 will occur at or beyond the rightmost j satisfying $\neg\phi_1$, so $w, i \models \phi_1$ **since** ϕ_2. In case 2, the rightmost j satisfying ϕ_2 occurs before the rightmost j satisfying $\neg\phi_1$, so $w, i \not\models \phi_1$ **since** ϕ_2. In case 3, there is no j satisfying ϕ_2, so $w, i \not\models \phi_1$ **since** ϕ_2.

So the idea is to attend to the rightmost position such that ϕ_1 is false or ϕ_2 is true. Then return whether at that position ϕ_2 is true or not. If there is no such position, return false.

Add (a trivial self-attention layer and) a FFNN (using Theorem 1.32) to compute

$$\mathbf{A}'[i] = \begin{bmatrix} \mathbb{I}[\mathbf{w}, i \models \phi_1] \\ \mathbb{I}[\mathbf{w}, i \models \phi_2] \\ \neg\mathbb{I}[\mathbf{w}, i \models \phi_1] \vee \mathbb{I}[\mathbf{w}, i \models \phi_2] \\ 0 \end{bmatrix}. \tag{2.17}$$

Then add a self-attention layer:

$$\mathbf{Q}[i] = \begin{bmatrix}1\end{bmatrix} \qquad \mathbf{K}[j] = \begin{bmatrix}\mathbb{I}[\neg(\mathbf{w}, j \models \phi_1) \lor (\mathbf{w}, j \models \phi_2)]\end{bmatrix} \tag{2.18}$$

$$\mathbf{V}[j] = \begin{bmatrix} 0 \\ 0 \\ 0 \\ \mathbf{w}, j \models \phi_2 \end{bmatrix}. \tag{2.19}$$

For any position i, the position j_i that receives attention is the rightmost one left of i that either satisfies ϕ_2 (in which case ϕ_1 must be satisfied from j_i to i exclusive) or does not satisfy ϕ_1 (in which case ϕ_2 must not be satisfied from j_i to i exclusive).

Then i satisfies (ϕ_1 **since** ϕ_2) if and only if j_i satisfies ϕ_2. So the value tests for whether ϕ_2 is satisfied.

There are two edge cases to consider. First, if there are no positions such that ϕ_1 is false or ϕ_2 is true, then all positions maximize the score. The winner is position $(i-1)$, and at that position, ϕ_2 is false, so the attention outputs false, which is correct. Second, at the very leftmost position ($i = 0$), there are no unmasked positions, so the attention outputs the zero vector (= false), which is correct. □

Since this construction uses only position-independent queries (0 or 1), a perhaps surprising consequence is that every transformer encoder with rightmost hard attention and strict future masking is equivalent to one that uses only position-independent queries.

2.4 Additional Results

The close correspondence between masked hard-attention transformers and LTL not only provides a characterization of the expressive power of (strictly masked unique hard-attention) transformers, but also gives fine-grained insights into what factors contribute to their expressive power. By direct application of known results about LTL, we can better understand the role played by strict masking, positional encodings, and depth.

2.4.1 Stutter-Invariance

The choice of strict masking deviates from standard practice, and may appear to be a deficiency, but in the setting of masked hard-attention transformers, it actually contributes to expressive power. Non-strictness is known to reduce expressivity in LTL [Peled and Wilke, 1997], and we can then show that it reduces expressivity in masked hard-attention transformers as well. Intuitively, non-strict masked operations are unable to distinguish between consecutive positions that have the same symbol. More formally, a language over Σ is called *stutter-invariant* iff for all $u, v \in \Sigma^*$ and $a \in \Sigma$, $uav \in L$ iff $uaav \in L$. An example of a language that is star-free but not stutter-invariant is $\{a\}$.

Theorem 2.7. *Masked hard-attention transformers with only non-strict masking recognize exactly the stutter-invariant star-free languages.*

Proof. Peled and Wilke [1997] prove that LTL with non-strict operators recognizes exactly the stutter-invariant star-free languages. Non-strict **since** is defined as follows:

$\mathbf{w}, i \models \phi_1$ **since** ϕ_2 if there is a $j \leq i$ such that $\mathbf{w}, j \models \phi_2$, and for all k such that $j < k \leq i$, we have $\mathbf{w}, k \models \phi_1$

The proofs of Theorems 2.1 and 2.5 may be adapted to use non-strict temporal operators and non-strict masking. Thus, non-strict masked hard-attention transformers and non-strict LTL are equivalent, and the result follows. □

2.4.2 Regular Languages in AC^0

So far we have only considered masked hard-attention transformers without position embeddings, but the proofs Theorems 2.1 and 2.5 go through for transformers extended with position embeddings with *finite image* (the set $\bigcup_n \{\mathrm{PE}_n(i) \mid i \in [n]\}$ is finite) and FO or LTL extended with additional monadic numerical predicates (predicates with one argument that depend only on n, not $\mathbf{w}$). Here, we consider the case of sinusoidal position embeddings, which are similar to the original position embedding [Vaswani et al., 2017].

For any even d, let us define a *rational sinusoidal positional embedding* with d dimensions to be a position embedding

$$\mathrm{PE}_n(i) = \begin{bmatrix} \sin(2\pi f_0 i) \\ \cos(2\pi f_0 i) \\ \cdots \\ \sin(2\pi f_{d/2-1} i) \\ \cos(2\pi f_{d/2-1} i) \end{bmatrix} \qquad f_0, \ldots, f_{d/2-1} \in \mathbb{Q}.$$

Admittedly, in the original definition, the f_k were not rational.

Corollary 2.8. *Masked hard-attention transformers with rational sinusoidal position embeddings recognize exactly the regular languages in* AC^0 *(that is, regular languages definable by a family of Boolean circuits with polynomial size and constant depth).*

Proof. Let MOD be the collection of predicates $\mathrm{MOD}^r_m(i)$ for all $0 \le r < m$, which hold just in case $i \equiv r \pmod{m}$. The regular languages in AC^0 are exactly the languages definable in $\mathsf{FO}[\mathrm{MOD}]$ or first-order logic with modular predicates [Barrington et al., 1992].

(Masked hard-attention transformers to $\mathsf{FO}[\mathrm{MOD}]$) Let PE_n be a sinusoidal positional embedding. Since the f's are rational, PE_n has finite image and is also periodic (that is, there is an integer m such that for all i, i and $i+m$ have the same embedding). So we can adapt the proof Theorem 2.1, which expresses a masked hard-attention transformer in FO, modifying Eq. (2.4) to

$$\mathrm{act}_{0,\mathbf{v}}(i) = \bigvee_{\substack{a \in \Sigma \\ r \in [m] \\ \mathrm{WE}(a)+\mathrm{PE}(r)=\mathbf{v}}} \left(Q_a(i) \wedge \mathrm{MOD}^r_m(i)\right). \tag{2.20}$$

Thus, transformers with positional embedding PE_n are contained in $\mathsf{FO}[<, \mathrm{MOD}]$, which are the regular languages in AC^0.

($\mathsf{FO}[\mathrm{MOD}]$ to $\mathsf{LTL}[\mathrm{MOD}]$) By $\mathsf{LTL}[\mathrm{MOD}]$, we mean LTL extended with MOD predicates as defined above. The proof of Kamp's theorem (Theorem 1.21) works with these (or any) additional predicates.

($\mathsf{LTL}[\mathrm{MOD}]$ to masked hard-attention transformers) We can use a 2-layer ReLU network to compute MOD^r_m [Chiang et al., 2023, Lemma 20]:

$$\begin{aligned} h(i) &= \mathrm{ReLU}\left(\sin 2\pi r/m \sin 2\pi i/m + \cos 2\pi r/m \cos 2\pi i/m - \cos 2\pi/m\right) \\ &= \mathrm{ReLU}(\cos(2\pi(i-r)/m)) \\ \mathrm{MOD}^r_m(i) &= (1 - \cos 2\pi/m) h(i). \end{aligned}$$

Thus $\mathsf{LTL}[\text{MOD}]$ is contained in the languages recognized by masked hard-attention transformers with rational sinusoidal position embeddings.

Thus, this class of transformers defines exactly the class of regular languages in AC^0. □

2.4.3 Depth

There is a close relationship between **since** operators and self-attention layers. In fact, the **since**-nesting depth of a formula corresponds to the number of attention layers in the corresponding transformer, assuming we use multi-headed attention layers. We've already defined the *depth* of a masked hard-attention transformer to be the number of attention layers it has (L).

The *temporal depth* of an LTL formula is defined inductively as follows:

$$\begin{aligned} \mathrm{dp}(Q_a) &= 0 & \mathrm{dp}(\phi \wedge \psi) &= \max(\mathrm{dp}(\phi), \mathrm{dp}(\psi)) \\ \mathrm{dp}(\neg\phi) &= \mathrm{dp}(\phi) & \mathrm{dp}(\phi \textbf{ since } \psi) &= \max(\mathrm{dp}(\phi), \mathrm{dp}(\psi)) + 1. \end{aligned}$$

Let $\mathsf{MUHAT}(\blacktriangleright F)_k$ be the languages recognizable by multi-head transformers of depth k using only future-masked rightmost-hard attention. Let LTL_k be the languages definable by LTL formulas of depth k.

The proof of Theorem 2.5 can be refined to show that an LTL formula of depth k can be simulated by a transformer of depth k. This requires the use of multi-head attention in order to perform more attention operations in parallel.

Theorem 2.9. *For all $k \geq 0$, $\mathsf{LTL}_k \subseteq \mathsf{MUHAT}(\blacktriangleright F)_k$.*

Proof. We will show that every formula up to depth k can be simulated by a transformer of depth k, by induction on k.

If ϕ is depth 0, it is a Boolean combination of the Q_a predicates. So it can be computed entirely in the word embeddings (cf. Eq. (2.13)):

$$tfr_\phi(\mathbf{w})[i] = \left[\mathbb{I}[\mathbf{w}, i \models \phi]\right]. \tag{2.21}$$

For the inductive step, first note that ϕ is a Boolean combination of formulas $\phi_0, \ldots, \phi_{m-1}$ where each ϕ_i has depth at most $(k+1)$ and is of the form $\phi_i = \phi_{i1}$ **since** ϕ_{i2}, so ϕ_{i1} and ϕ_{i2} have depth at most k. By the induction hypothesis, there are transformers simulating the ϕ_{i1} and ϕ_{i2}, and we can use the parallel composition lemma (Lemma 2.6) to combine them. We then add a multi-head self-attention layer to simulate each **since** operator in parallel, and then a feed-forward layer to simulate the Boolean combination. Thus, we can simulate ϕ with a transformer of depth k. □

Here, we simulated a masked hard-attention transformer in LTL indirectly, via simulation in FO and Kamp's theorem. However, we can also perform the simulation directly [Yang et al., 2023, Theorem 4], making it possible to relate the depth of the masked hard-attention transformer with the resulting LTL formula:

Corollary 2.10. *For all $k \geq 0$, $\mathsf{MUHAT}(\blacktriangleright F)_k \subseteq \mathsf{LTL}_{2k}$.*

Now that we've related the depth of masked hard-attention transformers with the depth of LTL formulas, we can apply known results about the LTL depth hierarchy in the setting of transformers. Let $\Sigma = \{\mathtt{a}, \mathtt{b}, \mathtt{c}\}$; then STAIR_k is the set of strings which, after deleting c's, contain $\mathtt{a}^k$ as a substring. For example, abaca is in STAIR_2 but not STAIR_3. It was shown by Etessami and Wilke [2000] that the STAIR language separates the levels of the LTL depth hierarchy:

Theorem 2.11. *For all* $k \geq 0$, $STAIR_{k+1} \in \mathsf{LTL}_{k+1} \setminus \mathsf{LTL}_k$. *Thus,* $\mathsf{LTL}_k \subsetneq \mathsf{LTL}_{k+1}$.

Using this, we can show the following:

Theorem 2.12. *For all* $k \geq 0$, $\mathsf{MUHAT}(\blacktriangleright F)_k \subsetneq \mathsf{MUHAT}(\blacktriangleright F)_{2k+1}$. *Thus, the depth hierarchy for masked hard-attention transformers is strict.*

Proof. By composing several results from above:

$$\mathsf{MUHAT}(\blacktriangleright F)_k \subseteq \mathsf{LTL}_{2k} \subsetneq \mathsf{LTL}_{2k+1} \subseteq \mathsf{MUHAT}(\blacktriangleright F)_{2k+1}.$$

□

So in contrast to feedforward neural networks, where two layers is enough to approximate any function, with masked hard-attention transformers, it's always possible to increase expressivity by adding more layers.

Chapter 3

Decoders with Intermediate Steps

Today, we are going to look at one of the two earliest results mentioned in Chapter 1:

> For any Turing machine M, there is a transformer decoder with average-hard attention and intermediate steps that simulates M.

We start by defining some key terms.

3.1 Definitions

3.1.1 Average-hard attention

As is common, this proof simplifies attention by making it focus attention only on the positions with the maximum score ($\mathbf{s}$). If there is more than one maximal position, attention is distributed evenly among them.

For any vector $\mathbf{x} \in \mathbb{R}^d$, define $M(\mathbf{x}) = \{i \in [n] \mid \forall j \in [n], \mathbf{x}[j] \leq \mathbf{x}[i]\}$ to be the set of indices of the maximal elements of $\mathbf{x}$. In *average*-hard attention, Eq. (1.24) is replaced with:

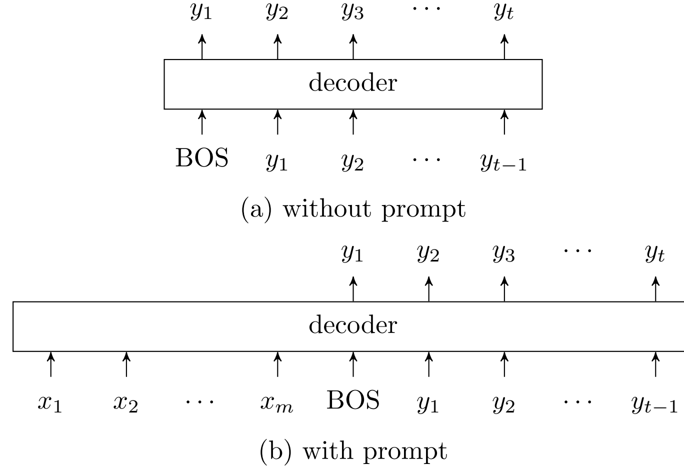
$$\alpha[i,j] = \frac{\mathbb{I}[j \in M(\mathbf{s}[i,:])]}{|M(\mathbf{s}[i,:])|}. \tag{3.1}$$

Average-hard attention was also called *hard* attention by Pérez et al. [2021] and *saturated* attention by Merrill et al. [2022], and has been argued to be a realistic approximation to how trained transformers behave in practice [Merrill et al., 2021].

3.1.2 Transformer decoders

A *transformer decoder* is a transformer encoder *tfr* with future masking in its attention, typically used to generate rather than recognize strings. GPT and its competitor LLMs are all transformer decoders.

We assume that Σ contains a special symbol BOS that does not occur anywhere else; later, we will add several other special symbols. The input string is the prefix of previously-generated symbols, $\mathbf{y}_{<t} = y_0 \cdots y_{t-1}$, where $y_0 = \text{BOS}$. The output is a



(a) without prompt

(b) with prompt

Figure 3.1: Generating strings from a transformer decoder.

probability distribution $\hat{p}(y_t \mid \mathbf{y}_{<t})$ over the next symbol,

$$\begin{aligned} \mathit{out} \colon \mathbb{R}^d &\to \mathbb{R} \\ \mathbf{x} &\mapsto \mathbf{W}\mathbf{x} + \mathbf{b} \end{aligned} \tag{3.2}$$

$$\hat{p}(\cdot \mid \mathbf{y}_{<t}) = \operatorname{softmax}(\mathit{out}(\mathit{tfr}(\mathbf{y}_{<t})[t-1])) \tag{3.3}$$

where parameters $\mathbf{W} \in \mathbb{R}^{|\Sigma| \times d}$ and $\mathbf{b} \in \mathbb{R}^{|\Sigma|}$.

To sample a string, we first sample y_1 from $\hat{p}(y_1 \mid \text{BOS})$, then, for each time step $t > 1$, sample y_t from $\hat{p}(y_t \mid \mathbf{y}_{<t})$. The process stops when $y_t = \text{EOS}$. Because each sampled output symbol becomes part of the input at the next time step, this kind of model is called *autoregressive*. See Fig. 3.1a.

In most (not all) theoretical papers about transformer decoders, we want the decoder to output a single next symbol instead of a probability distribution over next symbols. To do this, we can select the argmax of $\mathit{out}(\mathit{tfr}(\mathbf{y}_{<t})[t-1])$ instead. Warning: In general, selecting the argmax at each step does *not* give you the highest-probability string.

We can also provide a *prompt* $\mathbf{x}$ to the decoder, which the decoder can see as part of its input but doesn't have to output. In that case, for $t \geq 1$, the input string is $\mathbf{x} \cdot \mathbf{y}_{<t}$, and the output is y_t. See Fig. 3.1b.

3.1.3 Intermediate steps

In many applications of transformer decoders, the prompt $\mathbf{x}$ is some kind of question, and the desired output $\mathbf{y}$ is the answer. For example, $\mathbf{x} =$ `101*101` and $\mathbf{y} =$ `10201`. Researchers have found in practice that sometimes a transformer decoder isn't very good at answering certain kinds of questions, but if one allows the decoder to insert a number of *intermediate* time steps between the prompt and the final output, it sometimes performs much better. This is known as a *scratchpad* [Nye et al., 2022] or *chain of thought* [Wei et al., 2022]. Here, we're only interested in the case where the final output is a single symbol, so we have the following definition.

Definition 3.1. Let f be a transformer decoder. For any string $\mathbf{x} \in \Sigma^*$, we say that f, on prompt $\mathbf{x}$, outputs y_T after T intermediate steps if there is a string $\mathbf{y} = y_1 \cdots y_T$ such that for all $t = 1, \ldots, T$, the output distribution $f(\mathbf{x} \cdot \mathbf{y}_{<t})$ is maximized by y_t.

3.1.4 Turing machines

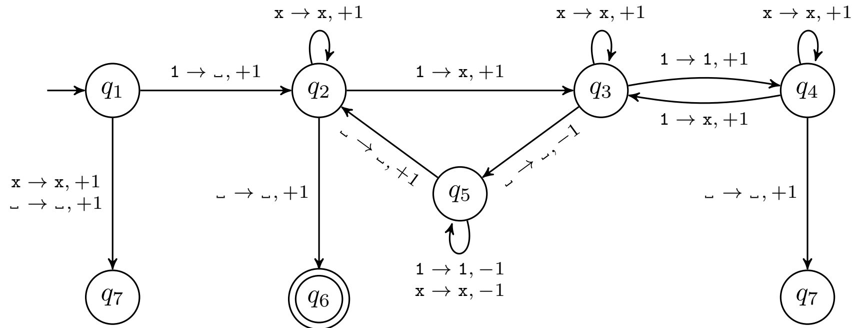
We assume that you are familiar with Turing machines. We use the definition of Turing machine in the textbook by Sipser [2013], with one small modification. Just to make sure we're on the same page, we give the barest of definitions here.

Definition 3.2. A Turing machine is a tuple $M = (Q, \Sigma, \Gamma, \delta, q_{\text{start}}, q_{\text{accept}}, q_{\text{reject}})$, where

- Q is a finite set of states
- Σ is a finite input alphabet, where $␣ \notin \Sigma$
- Γ is a finite tape alphabet, where $\Sigma \cup \{␣\} \subseteq \Gamma$
- $\delta \colon Q \times \Gamma \to Q \times \Gamma \times \{-1, +1\}$ is the transition function.

The tape has a left end and extends infinitely to the right. On input $\mathbf{w} \in \Sigma^*$, the tape is initialized to $\mathbf{w}␣␣\cdots$. If the current state is q, the current tape symbol is a, and $\delta(q, a) = (r, b, m)$, then the machine enters state r, writes a b, and moves left if $m = -1$, right if $m = +1$. If the machine enters state q_{accept}, it halts and accepts $\mathbf{w}$; if it enters state q_{reject}, it halts and rejects $\mathbf{w}$.

Example 3.3. Here's an example Turing machine [Sipser, 2013], with $q_{\text{start}} = q_1$, $q_{\text{accept}} = q_6$ (marked with a double circle), $q_{\text{reject}} = q_7$. It decides the language $\{\texttt{1}^{2^m} \mid m \geq 0\}$.



The reject state q_7 appears twice to reduce clutter.

The (not very exciting) run of this machine on string 11 is:

$$\begin{array}{ll}
q_1 & \underset{\wedge}{\texttt{1}}\texttt{1}␣\cdots \\
q_2 & ␣\underset{\wedge}{\texttt{1}}␣\cdots \\
q_3 & ␣\texttt{x}\underset{\wedge}{␣}\cdots \\
q_5 & ␣\underset{\wedge}{\texttt{x}}␣\cdots \\
q_5 & \underset{\wedge}{␣}\texttt{x}␣\cdots \\
q_2 & ␣\underset{\wedge}{\texttt{x}}␣\cdots \\
q_6 & \text{accept}
\end{array}$$

3.2 Transformers Simulating Turing Machines

Theorem 3.4. *For any Turing machine M with input alphabet Σ, there is a transformer decoder f with average-hard attention that is equivalent to M in the following sense. For any string $\mathbf{w} \in \Sigma^*$:*

- *If M halts and accepts on input $\mathbf{w}$, then there is a T such that f, on prompt $\mathbf{w}$, outputs* ACC *after T intermediate steps.*
- *If M halts and rejects on input $\mathbf{w}$, then there is a T such that f, on prompt $\mathbf{w}$, outputs* REJ *after T intermediate steps.*
- *If M does not halt on input $\mathbf{w}$, then there does not exist a T such that f, on prompt $\mathbf{w}$, outputs either* ACC *or* REJ.

The rest of this chapter proves the above theorem. There are several related proofs in the literature [Pérez et al., 2021, Bhattamishra et al., 2020b, Merrill and Sabharwal, 2024]; ours is an amalgam of these.

The key idea in most of these proofs is that a transformer has no memory to store the contents of the tape. All it has is the intermediate steps, which it uses to record *changes* to the contents of the tape. Whenever it needs to read a cell of the tape, it must use the record of changes to reconstruct the contents of the cell.

Let $M = (Q, \Sigma, \Gamma, \delta, q_{\text{start}}, q_{\text{accept}}, q_{\text{reject}})$. The alphabet of *f.tfr* is

$$\Sigma' = \Sigma \cup \{\text{BOS}, \text{ACC}, \text{REJ}\} \cup (Q \times \Gamma \times \{-1, +1\}).$$

At each time step starting with BOS, the network outputs a triple $(r, b, m) \in (Q \times \Gamma \times \{-1, +1\})$ indicating what the next simulated action of M is.

Each time step $i = 0, 1, \ldots$ of the transformer proceeds as follows.

1. Unpack the current input symbol, $x^{(i)}$:
   - If $x^{(i)} \in \Sigma \cup \text{BOS}$, let $q^{(i)} = \bot$ and $m^{(i-1)} = 0$.
   - Else, let $(q^{(i)}, b^{(i-1)}, m^{(i-1)}) = x^{(i)}$.
2. Compute the head position: $h^{(i)} = \sum_{j=0}^{i} m^{(j-1)}$.
3. Compute the symbol under the head, $a^{(i)}$:
   - Find j^*, the rightmost position $j < i$ such that $h^{(j)} = h^{(i)}$.
   - If j^* exists and $b^{(j)} \neq \bot$, let $a^{(i)} = b^{(j^*)}$.
   - Else, if $x^{(h^{(i)})} \in \Sigma$, let $a^{(i)} = x^{(h^{(i)})}$.
   - Else, let $a^{(i)} = \sqcup$.
4. Compute the next transition:
   - If $x^{(i)} \in \Sigma$, just output $y^{(i)} = x^{(i)}$. (It will be ignored anyway.)
   - Else, if $x^{(i)} = \text{BOS}$, let $q^{(i+1)} = q_{\text{start}}$, $b^{(i)} = a^{(i)}$, and $m^{(i)} = 0$.
   - Else, let $(q^{(i+1)}, b^{(i)}, m^{(i)}) = \delta(q^{(i)}, a^{(i)})$.
   - If $q^{(i+1)} = q_{\text{accept}}$ or q_{reject}, output $y^{(i)} = \text{ACC}$ or $y^{(i)} = \text{REJ}$, respectively.
   - Else, output $y^{(i)} = (q^{(i+1)}, b^{(i)}, m^{(i)})$.

For example, the run from Example 3.3 is simulated by:

i	tape	$x^{(i)}$	$q^{(i)}$	$b^{(i-1)}$	$m^{(i-1)}$	$h^{(i)}$	$a^{(i)}$	$y^{(i)}$
0		$\texttt{1}$	$\bot$	$\bot$	0	0	$\texttt{1}$	$\texttt{1}$
1		$\texttt{1}$	$\bot$	$\bot$	0	0	$\texttt{1}$	$\texttt{1}$
2		BOS	$\bot$	$\bot$	0	0	$\texttt{1}$	$(q_1, \texttt{1}, 0)$
3	$\underset{\wedge}{\texttt{1}}\texttt{1}\textvisiblespace\cdots$	$(q_1, \texttt{1}, 0)$	q_1	$\texttt{1}$	0	0	$\texttt{1}$	$(q_2, \textvisiblespace, +1)$
4	$\textvisiblespace\underset{\wedge}{\texttt{1}}\textvisiblespace\cdots$	$(q_2, \textvisiblespace, +1)$	q_2	$\textvisiblespace$	$+1$	1	$\texttt{1}$	$(q_3, \texttt{x}, +1)$
5	$\textvisiblespace\texttt{x}\underset{\wedge}{\textvisiblespace}\cdots$	$(q_3, \texttt{x}, +1)$	q_3	$\texttt{x}$	$+1$	2	$\textvisiblespace$	$(q_5, \textvisiblespace, -1)$
6	$\textvisiblespace\underset{\wedge}{\texttt{x}}\textvisiblespace\cdots$	$(q_5, \textvisiblespace, -1)$	q_5	$\textvisiblespace$	-1	1	$\texttt{x}$	$(q_5, \texttt{x}, -1)$
7	$\underset{\wedge}{\textvisiblespace}\texttt{x}\textvisiblespace\cdots$	$(q_5, \texttt{x}, -1)$	q_5	$\texttt{x}$	-1	0	$\textvisiblespace$	$(q_2, \textvisiblespace, +1)$
8	$\textvisiblespace\underset{\wedge}{\texttt{x}}\textvisiblespace\cdots$	$(q_2, \textvisiblespace, +1)$	q_2	$\textvisiblespace$	$+1$	1	$\texttt{x}$	ACC

Now we have to construct transformer layers to perform the above steps. Each input vector contains a word embedding, $e(x^{(i)})$, and a position embedding with 4 components:

$$\mathbf{A}^{(0)}[i] = \begin{bmatrix} e(x^{(i)}) \\ 1/(i+1) \\ 1 \\ i \\ i^2 \end{bmatrix}. \tag{3.4}$$

Step 1 is piecewise linear, so it can be computed by a FFNN (Theorem 1.33). Afterwards, the activation vector at position i is:

$$\mathbf{A}^{(1)}[i] = \begin{bmatrix} e(x^{(i)}) \\ 1/(i+1) \\ 1 \\ i \\ i^2 \\ e(q^{(i)}) \\ e(b^{(i-1)}) \\ m^{(i-1)} \end{bmatrix}. \tag{3.5}$$

Step 2 can be computed by a uniform self-attention layer:

$$\mathbf{Q}[i] = 0 \qquad \mathbf{K}[j] = 0 \qquad \mathbf{V}[j] = \begin{bmatrix} \mathbf{0} \\ m^{(i-1)} \end{bmatrix}. \tag{3.6}$$

Uniform self-attention computes an average, not a sum. Since at time step i, there are $i+1$ positions to average over, the result is $h^{(i)}/(i+1)$, not $h^{(i)}$. We'll correct this in the next step.

$$\mathbf{A}^{(2)}[i] = \begin{bmatrix} e(x^{(i)}) \\ 1/(i+1) \\ 1 \\ i \\ i^2 \\ e(q^{(i)}) \\ e(b^{(i-1)}) \\ m^{(i-1)} \\ h^{(i)}/(i+1) \end{bmatrix} \tag{3.7}$$

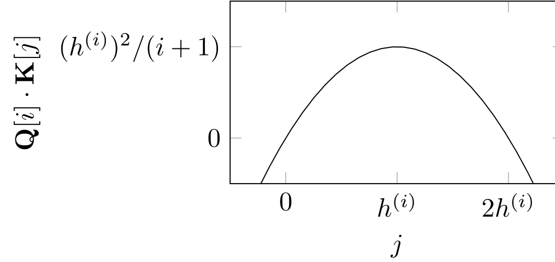
Step 3 is the most difficult step. There are several schemes that have been proposed for this. All of them use average-hard attention, and all of them further modify the transformer in some way: changing dot-product to something else [Pérez et al., 2021], adding components to the position embedding [Pérez et al., 2021, Barceló et al., 2024, Strobl et al., 2024a], or applying layer normalization only to selected components [Merrill and Sabharwal, 2024]. The construction here is closest to that of Strobl et al. [2024a].

It uses two self-attention layers. The first layer changes $h^{(i)}/(i+1)$ to $h^{(i)}$ by treating the position embeddings as a lookup table [Barceló et al., 2024].

$$\mathbf{Q}[i] = \begin{bmatrix} 2h^{(i)}/(i+1) \\ -1/(i+1) \end{bmatrix} \qquad \mathbf{K}[j] = \begin{bmatrix} j \\ j^2 \end{bmatrix} \qquad \mathbf{V}[j] = \begin{bmatrix} \mathbf{0} \\ j \\ j^2 \\ e(x^{(j)}) \end{bmatrix} \tag{3.8}$$

$$\mathbf{Q}[i] \cdot \mathbf{K}[j] = \frac{1}{i+1} j(2h^{(i)} - j). \tag{3.9}$$

Then the attention scores are uniquely maximized when $j = h^{(i)}$:



So the attention layer outputs $h^{(i)}$, and also some other quantities which we'll need shortly. The activation vector at position i is:

$$\mathbf{A}^{(3.5)}[i] = \begin{bmatrix} e(x^{(i)}) \\ 1/(i+1) \\ 1 \\ i \\ i^2 \\ e(q^{(i)}) \\ e(b^{(i-1)}) \\ m^{(i-1)} \\ h^{(i)}/(i+1) \\ h^{(i)} \\ (h^{(i)})^2 \\ e(x^{(h^{(i)})}) \end{bmatrix}. \tag{3.10}$$

Recall that we need to search positions $j < i$ such that $h^{(j)} = h^{(i)}$. But future-masked attention looks at positions $j \leq i$. To get around this, we search positions $j \leq i$ such that $h^{(j-1)} = h^{(i)}$. So we will need

$$h^{(i-1)} = h^{(i)} - m^{(i-1)} \tag{3.11}$$

$$(h^{(i-1)})^2 = (h^{(i)})^2 - 2h^{(i)}m^{(i-1)} + (m^{(i-1)})^2 \tag{3.12}$$

$$= \begin{cases} (h^{(i)})^2 + 2h^{(i)} + 1 & \text{if } m^{(i-1)} = -1 \\ (h^{(i)})^2 - 2h^{(i)} + 1 & \text{if } m^{(i-1)} = +1 \end{cases} \tag{3.13}$$

which can both be computed using a FFN. So

$$\mathbf{A}^{(4)}[i] = \begin{bmatrix} e(x^{(i)}) \\ 1/(i+1) \\ 1 \\ i \\ i^2 \\ e(q^{(i)}) \\ e(b^{(i-1)}) \\ m^{(i-1)} \\ h^{(i)}/(i+1) \\ h^{(i)} \\ (h^{(i)})^2 \\ e(x^{(h^{(i)})}) \\ h^{(i-1)} \\ (h^{(i-1)})^2 \end{bmatrix}. \tag{3.14}$$

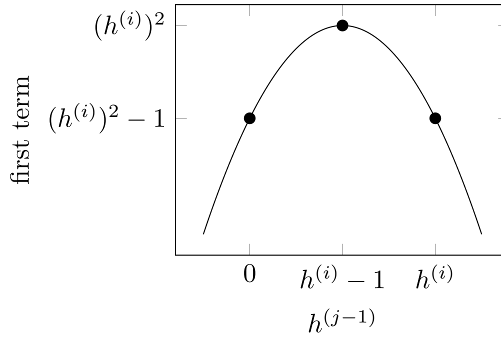
The second self-attention layer uses another variation of the lookup trick:

$$\mathbf{Q}[i] = \begin{bmatrix} 2h^{(i)} \\ -1 \\ \frac{1}{2(i+1)} \end{bmatrix} \qquad \mathbf{K}[j] = \begin{bmatrix} h^{(j-1)} \\ (h^{(j-1)})^2 \\ j \end{bmatrix} \qquad \mathbf{V}[j] = \begin{bmatrix} \mathbf{0} \\ h^{(j-1)} \\ e(b^{(j-1)}) \end{bmatrix} \tag{3.15}$$

$$\mathbf{Q}[i] \cdot \mathbf{K}[j] = \underbrace{h^{(j-1)}(2h^{(i)} - h^{(j-1)})}_{\text{find } h^{(j-1)} = h^{(i)}} + \underbrace{\frac{j}{2(i+1)}}_{\text{find rightmost}} \tag{3.16}$$

The first term is maximized when $h^{(j-1)} = h^{(i)}$, in which case it attains its maximum of $(h^{(i)})^2$.

If there is more than one such position j, then because of the second term, j^* is the rightmost such j. This second term has to be increasing in j, but it also has to be small enough that it cannot make the second-highest score as big as the highest score. Since $h^{(j-1)}$ is an integer, the difference between the first- and second-best values of the first term is 1:



So we make the second term $\frac{j}{2(i+1)} \leq \frac{1}{2} < 1$.

This gives

$$\mathbf{H}^{(4)}[i] = \begin{bmatrix} e(x^{(i)}) \\ 1/(i+1) \\ 1 \\ i \\ i^2 \\ e(q^{(i)}) \\ e(b^{(i-1)}) \\ m^{(i-1)} \\ h^{(i)}/(i+1) \\ h^{(i)} \\ (h^{(i)})^2 \\ e(x^{(h^{(i)})}) \\ h^{(i-1)} \\ (h^{(i-1)})^2 \\ h^{(j^*-1)} \\ e(b^{(j^*-1)}) \end{bmatrix}. \tag{3.17}$$

Finally, we can use the FFNN to set

$$a^{(i)} = \begin{cases} b^{(j^*-1)} & h^{(j^*-1)} = h^{(i)} \text{ and } b^{(j^*-1)} \neq \bot \\ x^{(h^{(i)})} & \text{otherwise.} \end{cases} \tag{3.18}$$

So the activation vectors are:

$$\mathbf{A}^{(5)}[i] = \begin{bmatrix} e(x^{(i)}) \\ 1/(i+1) \\ 1 \\ i \\ i^2 \\ e(q^{(i)}) \\ e(b^{(i-1)}) \\ m^{(i-1)} \\ h^{(i)}/(i+1) \\ h^{(i)} \\ (h^{(i)})^2 \\ e(x^{(h^{(i)})}) \\ h^{(i-1)} \\ (h^{(i-1)})^2 \\ h^{(j^*-1)} \\ e(b^{(j^*-1)}) \\ e(a^{(i)}) \end{bmatrix}. \tag{3.19}$$

Step 4 is piecewise linear, so it can be computed by a FFNN (Theorem 1.33).

3.3 Open Questions

We don't know whether a similar construction exists for softmax-attention transformers.

Chapter 4

Encoders with Soft Attention

Today, we consider transformers that use softmax attention (as they do in practice). This seems to be the most difficult case to pin down. There are a number of upper bounds (transformers only recognize languages in some complexity class) and lower bounds (transformers can recognize any language in some complexity class), but no exact characterizations yet.

Here, we will look at one upper bound and one lower bound. Both of them are based on logics we've seen already, but extend them to allow *counting* and *arithmetic*.

4.1 Upper Bound

Upper bounds seem to require assuming some limitation on attention or numeric precision, or both. Strobl [2023] showed that average-hard attention transformers (AHATs, which we will encounter later) with $O(\log n)$-bit floating-point numbers only recognize languages in L (log-space) uniform TC^0. Here, we'll present the result of Merrill and Sabharwal [2023a] that SMATs with $O(\log n)$-bit floating-point numbers can only recognize languages in $\mathsf{DLOGTIME}$-uniform TC^0, which is equivalent to first-order logic with counting quantifiers, addition, and multiplication.

4.1.1 Arithmetic predicates

We can increase the expressivity of FO by adding more predicates besides $<$. The logic $\mathsf{FO}[+, \times]$ extends FO with predicates

$$\begin{aligned} \mathbf{w}, I &\models \mathrm{ADD}(x, y, z) && \text{if } I(x) + I(y) = I(z) \\ \mathbf{w}, I &\models \mathrm{MUL}(x, y, z) && \text{if } I(x)I(y) = I(z) \end{aligned}$$

For readability, we usually write $x + y = z$ in place of $\mathrm{ADD}(x, y, z)$, and similarly for other arithmetic operations. You could think of $x + y$ as an term that is interpreted as a natural number, but note that $\exists z. x + y = z$ may be false because z must be interpreted in $[n]$.

Exercise 4.1. Write formulas

1. $\mathrm{SUB}(x, y, z)$ such that

$$\mathbf{w}, I \models \mathrm{SUB}(x, y, z) \qquad \text{if } I(x) - I(y) = I(z)$$

2. $\text{DIV}(x, y, q, r)$ such that

$$\mathbf{w}, I \models \text{DIV}(x, y, q, r) \qquad \text{if } I(x) = I(y)I(q) + I(r)$$

Theorem 4.2. *The following formulas are definable in* $\mathsf{FO}[\text{BIT}]$*:*

(a) $\text{POW}(x, y, z)$ *iff* $z = x^y$

(b) $\text{BIT}(x, y, z)$ *iff the* y*-th bit in the binary representation of* x *is* z *[Immerman, 1999, Theorem 1.17.2]*

It is also possible (and in fact more common) to make BIT the built-in predicate, and to define in $\mathsf{FO}[\text{BIT}]$ predicates ADD and MUL.

Example 4.3. The following formula of $\mathsf{FO}[+, \times]$ tests whether a number is odd:

$$\text{ODD}(x) = \neg(\exists y. \text{ADD}(y, y, x)) = \neg(\exists y. y + y = x). \tag{4.1}$$

Previously we were unable to state a nice correspondence between FO and AC^0, but now we can:

Theorem 4.4 (Barrington et al., 1990)**.** $\mathsf{FO}[+, \times]$ *defines exactly the languages in* $\mathsf{DLOGTIME}$*-uniform* AC^0.

4.1.2 Uniform TC^0

We previously saw AC^k and NC^k (Definition 1.26); now we introduce TC^k, which has long been studied in connection with neural networks.

Definition 4.5. TC^k is the class of languages that can be recognized by families of circuits with unbounded fan-in, $O(\text{poly}(n))$ size, and $O((\log n)^k)$ depth, and have MAJORITY gates, which output 1 iff at least half of their inputs are 1.

$\mathsf{DLOGTIME}$-uniform TC^0 contains a lot of languages, but there are also a number of NC^1-complete languages that, under the widely-believed assumption that $\mathsf{TC}^0 \neq \mathsf{NC}^1$, do not belong to TC^0. Consequently, we don't think that transformers can recognize them either.

- One example of an NC^1-complete language is the Boolean Formula Value Problem (BFVP). The instances are propositional formulas built up from constants 0 and 1 and the connectives $\wedge$, $\vee$, $\neg$, and the problem is to decide whether such a formula is true or not. In other words, it is defined by the following context-free grammar:

$$\begin{aligned}
S &\rightarrow F_1 \\
F_1 &\rightarrow (F_1 \wedge F_1) \\
&\quad \mid (F_0 \vee F_1) \mid (F_1 \vee F_0) \mid (F_1 \vee F_1) \\
&\quad \mid (\neg F_0) \\
&\quad \mid 1 \\
F_0 &\rightarrow (F_0 \wedge F_0) \mid (F_0 \wedge F_1) \mid (F_1 \wedge F_0) \\
&\quad \mid (F_1 \vee F_1) \\
&\quad \mid (\neg F_1) \\
&\quad \mid 0
\end{aligned}$$

Linguistically, the ability to evaluate Boolean formulas is directly relevant to computations underlying compositional semantics. Indeed, Boole's original motivation was to assert that such descriptions codify a language of thought. Modern semantic theory, influenced more by the Lambda calculus thanks to work by Montague, Partee, and others, is a direct consequence. The relationship between neural networks and compositional behavior is a fraught one, and the subject of decades of debate from figures like Fodor, Pylyshyn, Smolensky, and others.

- The canonical example of a regular but NC^1-complete language is the word problem for S_5. A permutation of $[k]$ is a bijection $\pi\colon [k] \to [k]$, and S_k is the set of all permutations of $[k]$. Treating S_k as an alphabet and compositions of permutations as strings, we can define the language $\mathsf{W}(S_k)$ of compositions of permutations of $[k]$ that equal the identity permutation. For example, in S_3, the permutation (120) maps $0 \mapsto 1$, $1 \mapsto 2$, and $2 \mapsto 0$, so that $\mathsf{W}(S_3)$ contains $(120) \circ (120) \circ (120)$ but not $(120) \circ (120)$. These languages are easy for finite automata to recognize, but difficult with only fixed computation depth.

  The languages $\mathsf{W}(S_k)$ have some relevance to natural language: they resemble expressions like *the child of the enemy of Ann* where the interpretation of *the child of* is (roughly) a permutation of possible referents [Paperno, 2022], and problems that have been used to benchmark transformers' state-tracking abilities [Kim and Schuster, 2023].

4.1.3 Counting quantifiers

FOC is first order logic with *counting terms* [van Benthem and Icard, 2023].

Example 4.6. The majority language,

$$\mathsf{MAJORITY} = \{\mathbf{w} \in \{\mathtt{0}, \mathtt{1}\}^* \mid \mathbf{w} \text{ has more } \mathtt{1}\text{'s than } \mathtt{0}\text{'s}\}. \tag{4.2}$$

can be defined by the FOC formula

$$\underbrace{(\#z.Q_{\mathtt{0}}(z))}_{\text{number of } \mathtt{0}\text{'s}} < \underbrace{(\#z.Q_{\mathtt{1}}(z))}_{\text{number of } \mathtt{1}\text{'s}}. \tag{4.3}$$

The syntax of FOC is:

$$t ::= x \mid \#x.\phi_1 \tag{4.4}$$
$$\phi ::= Q_\sigma(t_1) \qquad \sigma \in \Sigma \tag{4.5}$$
$$\mid t_1 = t_2 \mid t_1 < t_2 \tag{4.6}$$
$$\mid \phi_1 \wedge \phi_2 \mid \phi_1 \vee \phi_2 \mid \neg\phi_1 \tag{4.7}$$
$$\mid \forall x.\phi_1 \mid \exists x.\phi_1 \tag{4.8}$$

We extend the definition of free variables (Eq. (1.5)) with:

$$\mathrm{FV}(x) = \{x\} \tag{4.9}$$
$$\mathrm{FV}(Q_\sigma(t_1)) = \mathrm{FV}(t_1) \qquad \sigma \in \Sigma \tag{4.10}$$
$$\mathrm{FV}(t_1 = t_2) = \mathrm{FV}(t_1) \cup \mathrm{FV}(t_2) \tag{4.11}$$
$$\mathrm{FV}(t_1 < t_2) = \mathrm{FV}(t_1) \cup \mathrm{FV}(t_2) \tag{4.12}$$
$$\mathrm{FV}(\#x.\phi_1) = \mathrm{FV}(\phi_1) \setminus \{x\} \tag{4.13}$$

Terms are interpreted as integers, as follows:

$$x^{\mathbf{w},I} = I(x) \tag{4.14}$$

$$(\#x.\phi_1)^{\mathbf{w},I} = |\{i \mid \mathbf{w}, I[x \mapsto i] \models \phi_1\}| \tag{4.15}$$

And we extend the definition of $\models$ (Eq. (1.9)) with:

$$\mathbf{w}, I \models Q_\sigma(t_1) \qquad \text{if } w_{t_1^{\mathbf{w},I}} = \sigma \tag{4.16}$$

$$\mathbf{w}, I \models t_1 = t_2 \qquad \text{if } t_1^{\mathbf{w},I} = t_2^{\mathbf{w},I} \tag{4.17}$$

$$\mathbf{w}, I \models t_1 < t_2 \qquad \text{if } t_1^{\mathbf{w},I} < t_2^{\mathbf{w},I} \tag{4.18}$$

FOC is sometimes defined using counting *quantifiers* [Immerman, 1999, p. 185–187], but the formulation above is equivalent and (we think) easier to use.

There is another logic called FOM, which is first-order logic with *majority quantifiers*. FOC and FOM are equivalent [Lange, 2004], and because FOM is more well-known, we'll also often refer to the class of languages that they both recognize as FOM.

The ability to count becomes more interesting when we can do something with counts other than compare them. Addition is actually already definable in FOC and FOM [Lange, 2004], so introducing + doesn't increase its expressivity, but introducing $\times$ does.

Threshold circuits and majority/counting in first-order logic are related by the following:

Theorem 4.7 (Barrington et al., 1990)**.** FOM[$\times$] *defines exactly the languages in* DLOGTIME*-uniform* TC^0.

And here's an example of something that counting quantifiers make possible (proof omitted):

Theorem 4.8 (Addition of $O(n)$ numbers with $O(\log n)$ bits)**.** *If $\phi(i,x)$ is a formula of* FOM[$\times$] *such that for each $i \in [n]$, $\phi(i,x)$ is true for exactly one x, then there is a formula* $\text{SUM}_\phi(y)$ *which is true iff*

$$y = \sum_{\substack{i \in [n] \\ x \text{ s.t. } \phi(i,x) \text{ true}}} x.$$

4.1.4 Precision

One key issue is that while unique-hard and average-hard attention (which we will encounter later) only produce rational numbers, soft attention produces real numbers. So far, attempts to obtain upper bounds on the expressivity of soft-attention transformers involve limiting the precision of the numbers involved.

Actual computers, of course, use floating-point numbers with a constant number of bits – usually 16 or 32. But Merrill and Sabharwal [2023a] argue that in $O(1)$ precision, attention cannot attend uniformly to a string of length n, because for large enough n, the attention weights (α) would all round down to zero. Instead, they use $O(\log n)$ bits of precision. Specifically, they use floating-point numbers, of the form $m \cdot 2^e$, where the mantissa m has $O(\log n)$ bits including a sign bit, and the exponent e has $O(\log n)$ bits including a sign bit. Our definition is slightly different from theirs:

Definition 4.9. A floating-point number with p bits (where p is even) is a pair (m, e) where m, e are integers in $[-2^{p/2-1}, 2^{p/2-1})$. Its value is $m \cdot 2^e$.

4.1.5 Main result

Theorem 4.10 (Merrill and Sabharwal, 2023a)**.** *For any $O(\log n)$-precision transformer encoder T that recognizes a language L, there is a formula of* $\mathsf{FOM}[\times]$ *that defines L.*

Proof. Merrill and Sabharwal [2023a]'s proof converted T to a family of threshold circuits, but we show how to go straight to $\mathsf{FOM}[\times]$.

Transformers only use a handful of operations: addition, multiplication, division, max, exp, and iterated addition. It suffices to show that these operations can be defined in $\mathsf{FOM}[\times]$ on $O(\log n)$-bit floating-point numbers.

Addition and *multiplication*, already defined on integers in Theorem 4.2, are generalized to $c \log n$ bit integers (where $c > 1$) by Schweikardt [2005, Theorem 3.4bd]. Then floating-point addition and multiplication can be defined using the following facts:

$$(m_1 \cdot 2^{e_1}) + (m_2 \cdot 2^{e_2}) = \begin{cases} (m_1 + m_2 \cdot 2^{e_2 - e_1}) \cdot 2^{e_1} & e_1 \geq e_2 \\ (m_1 \cdot 2^{e_1 - e_2} + m_2) \cdot 2^{e_2} & e_1 \leq e_2 \end{cases} \tag{4.19}$$

$$(m_1 \cdot 2^{e_1}) \cdot (m_2 \cdot 2^{e_2}) = m_1 m_2 \cdot 2^{e_1 + e_2}. \tag{4.20}$$

In all of the above, to get mantissas to be integers with the right number of bits, some rounding may be necessary. *Division* can be defined in terms of multiplication.

Iterated addition on floating-point numbers is more difficult:

$$\sum_{i=0}^{n-1} (m_i \cdot 2^{e_i}) = \left(\sum_{i=0}^{n-1} \underbrace{(m_i \cdot 2^{e_i - e})}_{(*)} \right) \cdot 2^e \tag{4.21}$$

where $(*)$ is rounded off to the nearest integer. The problem is that if some of the m_i are negative, the sum could end up much smaller than the largest summand. For example, suppose mantissas have 50 bits, and we want to compute

$$1 \cdot 2^0 + -1 \cdot 2^0 + 1 \cdot 2^{-100} = 1 \cdot 2^{-100}.$$

If we choose e to be the maximum of the e_i, then $1 \cdot 2^{-100}$ would round off to 0, giving a sum of 0. (This is known as *catastrophic cancellation.*) Instead, to make the sum exact, Merrill and Sabharwal [2023b] choose e to be the *minimum* of the e_i, which makes each $(*)$ into a $O(\text{poly}(n))$-bit integer. Iterated addition of these so-called long integers is still possible in $\mathsf{FOM}[\times]$ [Barrington and Maciel, 2000, Lecture 7]. (But if we had started with $O(n)$ bits, we would at this point have an exponential number of bits, so we'd need a different trick.)

For the *exponential function* ($\exp x$), first observe that

$$\begin{aligned} \exp x &= \exp_2(x/\log 2) && (4.22) \\ &= \exp_2(\lfloor x/\log 2 \rfloor) \exp_2(x/\log 2 - \lfloor x/\log 2 \rfloor) && (4.23) \\ &= \exp_2(\lfloor x/\log 2 \rfloor) \exp(\underbrace{x - \lfloor x/\log 2 \rfloor \log 2}_{r}). && (4.24) \end{aligned}$$

The first factor is just an integer power of 2. The second factor still involves exp, but now we know that $0 \leq r < \log 2$, which is small enough that $\exp r$ can be approximated by a truncated Taylor series (Merrill, p.c.; Hesse et al., 2002, Corollary 6.5). Let

$p \in O(\log n)$ be the number of bits of precision. Then we take the first p terms of the Taylor series about 0:

$$\exp r = \sum_{i=0}^{\infty} \frac{1}{i!} r^i = \sum_{i=0}^{p-1} \frac{1}{i!} r^i + R_p \tag{4.25}$$

where the Lagrange remainder term R_p is, for some z in $(0, r)$,

$$R_p = \frac{\exp z}{p!} r^p < \frac{\exp r}{p!} r^p < \frac{2}{p!} r^p \leq \frac{2}{2^p} r^p < \frac{1}{2^{p-1}}. \tag{4.26}$$

This means that our approximation has an error of at most "1 ulp" (unit in the last place), typical for floating-point library implementations. (CUDA guarantees an error of at most 2 ulp.)

So we compute Eq. (4.25) sans the remainder term R_p. Each term is an iterated product of $O(p) = O(\log n)$ numbers, which can be expressed in $\mathsf{FO}[+, \times]$ [Hesse et al., 2002, Theorem 5.1], and the summation of $p \in O(\log(n))$ terms can also be expressed in $\mathsf{FO}[+, \times]$ [Immerman, 1999]. □

4.2 Lower bound

With unique-hard attention, we were able to show an exact equivalence to FO and LTL. But softmax attention is trickier.

- Bhattamishra et al. [2020a] showed that one-state Parikh automata can be simulated by SMATs.
- Chiang et al. [2023] defined a logic called $\mathsf{FOC}[+; \mathrm{MOD}]$ and showed that it can be simulated by SMATs.
- Barceló et al. [2024] defined an extension of LTL with counting, called $\mathsf{LTL}[\#, +]$, and showed that it can be simulated by AHATs.
- Perhaps surprisingly, there isn't a published proof that softmax-attention transformers can simulate LTL (but we're working on it).

Here, we show that softmax-attention transformers can simulate a temporal logic without **since** but with a counting operator [Yang and Chiang, 2024]. We call this logic $\mathsf{K}_t[\#, +]$.

4.2.1 $\mathsf{K}_t[\#, +]$

The syntax of $\mathsf{K}_t[\#, +]$ is defined as follows:

$$t ::= \#[\phi_1] \tag{4.27}$$
$$\mid t_1 + t_2 \tag{4.28}$$
$$\phi ::= Q_\sigma \qquad \sigma \in \Sigma \tag{4.29}$$
$$\mid \phi_1 \wedge \phi_2 \mid \neg \phi_1 \tag{4.30}$$
$$\mid t_1 = t_2 \mid t_1 < t_2 \tag{4.31}$$

Other operators ($\vee$, $\rightarrow$, $>$, $\leq$, $\geq$) can be defined in terms of the ones above.

Terms are interpreted as integers. If t is a term, we write its interpretation with respect to string $\mathbf{w}$ and position i as $t^{\mathbf{w},i}$, defined as follows.

$$\#[\phi_1]^{\mathbf{w},i} = |\{j \le i \mid \mathbf{w}, j \models \phi_1\}| \tag{4.32}$$

$$(t_1 + t_2)^{\mathbf{w},i} = t_1^{\mathbf{w},i} + t_2^{\mathbf{w},i} \tag{4.33}$$

And we define the semantics as follows:

$$\mathbf{w}, i \models Q_\sigma \quad \text{iff } \mathbf{w}[i] = \sigma \tag{4.34}$$

$$\mathbf{w}, i \models \phi_1 \wedge \phi_2 \quad \text{iff } \mathbf{w}, i \models \phi_1 \text{ and } \mathbf{w}, i \models \phi_2 \tag{4.35}$$

$$\mathbf{w}, i \models \neg\phi \quad \text{iff } \mathbf{w}, i \not\models \phi \tag{4.36}$$

$$\mathbf{w}, i \models t_1 = t_2 \quad \text{iff } t_1^{\mathbf{w},i} = t_2^{\mathbf{w},i} \tag{4.37}$$

$$\mathbf{w}, i \models t_1 < t_2 \quad \text{iff } t_1^{\mathbf{w},i} < t_2^{\mathbf{w},i} \tag{4.38}$$

Unlike Barceló et al. [2024]'s $\mathsf{LTL}[\#, +]$, we do not have formulas $P(t)$ where P is a predicate other than $=$ or $<$.

Example 4.11. Below are some example $\mathsf{K}_t[\#, +]$ formulas and the languages they define:

Language	Formula
$\mathtt{a}^*\mathtt{b}^*$	$\#[Q_\mathtt{a} \wedge (\#[Q_\mathtt{b}] \ge 1)] = 0$
$\mathtt{a}^*\mathtt{b}^*\mathtt{a}^*$	$\#[Q_\mathtt{b} \wedge \#[Q_\mathtt{a} \wedge (\#[Q_\mathtt{b}] \ge 1)] \ge 1] = 0$
$\mathtt{a}^n\mathtt{b}^n\mathtt{c}^n$	$\#[Q_\mathtt{b} \wedge (\#[Q_\mathtt{c}] = 0)] = \#[Q_\mathtt{b}]$ $\wedge \#[Q_\mathtt{a} \wedge (\#[Q_\mathtt{b} \vee Q_\mathtt{c}] = 0)] = \#[Q_\mathtt{a}]$ $\wedge \#[Q_\mathtt{a}] = \#[Q_\mathtt{b}] \wedge \#[Q_\mathtt{b}] = \#[Q_\mathtt{c}] \wedge \#[Q_\mathtt{c}] = \#[Q_\mathtt{a}]$
Dyck-1	$(\#[Q_\mathtt{(}] = \#[Q_\mathtt{)}]) \wedge (\#[\#[Q_\mathtt{)}] > \#[Q_\mathtt{(}]] = 0)$
`hello`	$\#[\top] = 5 \wedge Q_\mathtt{o} \wedge \#[Q_\mathtt{l} \wedge \#[Q_\mathtt{e} \wedge \#[Q_\mathtt{h}] = 1] = 1] = 2$

4.2.2 Boolean and Count Representations

Many proofs of transformer lower bounds ignore the effects of layer normalization (Section 1.4.4). Here, layer normalization is actually a key part of the construction, so we will treat it with care.

First, we will ensure that the mean of every vector is zero, so that layer normalization does not add or subtract anything. Second, we will design the transformer so that if layer normalization scales a vector, it has no effect on the result of the computation. To help us keep track of any scaling, we initially ensure that the word/position embedding has as its 0th and 1st coordinates

$$\begin{bmatrix} 1 \\ -1 \end{bmatrix}.$$

Each vector contains Boolean values and counts. Instead of representing Boolean values as $\{0, 1\}$, we use the following zero-mean representations:

$$\text{true} : \begin{bmatrix} -1 \\ 1 \end{bmatrix} \qquad \text{false} : \begin{bmatrix} 1 \\ -1 \end{bmatrix}.$$

Similarly, to represent the integer C in position i, we use

$$\begin{bmatrix} \frac{C}{i+1} \\ -\frac{C}{i+1} \end{bmatrix}.$$

The input is a string of symbols as usual, but we require a BOS token to be prepended to the beginning of the input (or else we require that $1/(i+1)$ be in the position embedding of i).

Let $\mathbf{A} \in (\mathbb{R}^d)^*$ be a sequence of activation vectors (cf. Eqs. (1.42) and (1.43)). Assume that all subformulas and subterms of ϕ are numbered uniquely (that is, if ϕ_k is a subformula and $C_{k'}$ is a subterm, then $k \neq k'$). Each subformula ϕ_k is stored as two elements of $\mathbf{A}^{(i)}$. But at position 0, we always store a false value. Writing $\phi_k(i)$ as shorthand for $\mathbb{I}[\phi_k(i)]$:

$$\begin{aligned} \mathbf{A}[0, 2k : 2k+1] &= \begin{bmatrix} 1 \\ -1 \end{bmatrix} \\ \mathbf{A}[i, 2k : 2k+1] &= \begin{bmatrix} -2\phi_k(i)+1 \\ 2\phi_k(i)-1 \end{bmatrix} \qquad i > 0. \end{aligned} \tag{4.39}$$

Similarly, each count term C_k is stored as:

$$\begin{aligned} \mathbf{A}[0, 2k : 2k+1] &= \begin{bmatrix} 0 \\ 0 \end{bmatrix} \\ \mathbf{A}[1, 2k : 2k+1] &= \begin{bmatrix} -\frac{C_k(i)}{i+1} \\ \frac{C_k(i)}{i+1} \end{bmatrix} \qquad i > 0. \end{aligned} \tag{4.40}$$

The division of $C_k(i)$ by $(i+1)$ is a consequence of the fact that attention computes an average rather than a sum. Dealing with these divisions is a common feature of many transformer constructions. In contrast to other constructions that undo the divisions using nonstandard embeddings [Pérez et al., 2021, Barceló et al., 2024] or nonstandard versions of layer normalization [Merrill and Sabharwal, 2024], our construction uses no position embeddings and only standard layer normalization.

4.2.3 Counting

Counting is one of the important primitive operations that a transformer can perform. In the following, we show how to simulate a # term in $\mathsf{K}_t[\#, +]$ using a uniform attention layer.

Lemma 4.12. *Let $\mathbf{A}[*, 2k : 2k+1]$ store a sequence of Boolean values $\phi(i)$ as defined above. For any i, let $C(i)$ be the number of positions $j \leq i$ such that $\mathbf{A}[j, 2k : 2k+1]$ is true. Then there is a transformer block that computes, at each position i, and in two other dimensions $2k', 2k'+1$, the values $-\frac{C(i)}{i+1}$ and $\frac{C(i)}{i+1}$.*

Proof. We are given that

$$\mathbf{A}^{(\ell)}[i] = \begin{bmatrix} \vdots \\ -2\phi(i)-1 \\ 2\phi(i)+1 \\ \vdots \end{bmatrix}.$$

We want to simulate the counting term $\#[\phi(i)]$, that is, to compute $\pm\frac{C(i)}{i+1}$ in some other dimensions $2k', 2k'+1$. We construct a single transformer block. The self-attention, at each position i, uses uniform attention to compute the average of all

values up to and including position i in dimension $2k : 2k+1$:

$$\mathbf{H}^{(\ell+1)}[i] = \begin{bmatrix} \vdots \\ -2\phi(i) - 1 \\ 2\phi(i) + 1 \\ \vdots \\ -\frac{2}{i+1}\sum_i \phi(i) - 1 \\ \frac{2}{i+1}\sum_i \phi(i) + 1 \\ \vdots \end{bmatrix}.$$

Note, however, that instead of the desired value $\frac{C(i)}{i+1}$, we have actually computed $2\frac{C(i)}{i+1} - 1$, but it is straightforward to construct a FFNN that corrects this, giving

$$\mathbf{A}^{(\ell+1)}[i] = \begin{bmatrix} \vdots \\ -2\phi(i) - 1 \\ 2\phi(i) + 1 \\ \vdots \\ -\frac{1}{i+1}\sum_i \phi(i) \\ \frac{1}{i+1}\sum_i \phi(i) \\ \vdots \end{bmatrix}.$$

□

As a final note, recall that the layer normalization that occurs after the attention layer will apply a scaling factor to each column of the tensor. However, since we only compare values within the same column, this scaling factor will not affect the correctness of the computation, as we will describe in the following section.

4.2.4 Linear constraints

$\mathsf{K}_t[\#, +]$ can express any linear constraint on counts, that is, constraints of the form

$$\sum_{k \in K} a_k C_k(i) \geq 0 \tag{4.41}$$

where the C_k are count terms, the a_k are integer coefficients, and K is a finite set of indices. (The syntax of $\mathsf{K}_t[\#, +]$ allows other forms of constraints, but they can all be normalized into the above form.)

Lemma 4.13. *Let* $\mathbf{A} \in (\mathbb{R}^d)^*$ *be a sequence of* n *vectors in which, for each* $i \in [n]$ *and* $k \in K$, $\mathbf{A}[i, 2k : 2k+1]$ *stores a count* $C_k(i)$ *(using the representation in Eq.* (4.40)*). Let* a_k *for* $k \in K$ *be integer coefficients as in Eq.* (4.41)*. Let dimensions* $2k', 2k'+1$ *hold the value* 0 *across all positions. Then there is a stack of transformer blocks that computes, at each position* i*, and in two other dimensions* $2k', 2k'+1$*, whether the constraint Eq.* (4.41) *is true or false (using the representation in Eq.* (4.39)*).*

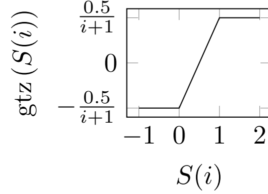
Proof. First, we will need the quantity $\frac{1}{i+1}$, which we obtain by uniformly attending to all positions, with a value of 1 for BOS and 0 for all other symbols.

Second, we use a FFNN to compute, at each position i, the linear combination

$$S(i) = \frac{\sum_{k\in K} a_k C_k(i)}{i+1} = \sum_{k\in K} a_k \frac{C_k(i)}{i+1}.$$

To test whether this is nonnegative, we construct a feed-forward layer that computes the function, given any input $S(i)$:

$$\operatorname{gtz}\left(S(i), \frac{1}{i+1}\right) = \min\left(\frac{0.5}{i+1}, \frac{S(i)}{i+1} - \frac{0.5}{i+1}\right) - \min\left(0, \frac{S(i)}{i+1}\right)$$



(This is where $\frac{1}{i+1}$ gets used, and must scale with the rest of the vector.)

Observe that $\operatorname{gtz}(\sum_k C_k(i) + 0.5)$ equals $\frac{0.5}{i+1}$ if $\sum_k C_k(i) \geq 0$, and $-\frac{0.5}{i+1}$ otherwise. This is because the counts must be integers, so if $\sum_k C_k(i) \geq 0$, then $\sum_k C_k(i) + 0.5 \geq 0.5$, and the expression will evaluate to $\frac{0.5}{i+1}$. Otherwise, $\sum_k C_k(i) \leq -1$, so $\sum_k C_k(i) + 0.5 < -0.5$, and the expression will evaluate to $-\frac{0.5}{i+1}$.

Both the linear combination and comparison with 0 can be packed into a single FFNN, and this FFNN can apply gtz to every other dimension too:

$$f\left(\begin{bmatrix} v_{i,0} \\ -v_{i,0} \\ \vdots \\ 0 \\ 0 \\ \vdots \\ v_{i,d/2-1} \\ -v_{i,d/2-1} \end{bmatrix}\right) = \begin{bmatrix} \operatorname{gtz}(v_{i,0}) \\ -\operatorname{gtz}(v_{i,0}) \\ \vdots \\ \operatorname{gtz}\left(\sum_{k\in K} a_k C_k(i)\right) \\ -\operatorname{gtz}\left(\sum_{k\in K} a_k C_k(i)\right) \\ \vdots \\ \operatorname{gtz}(v_{i,d/2-1}) \\ -\operatorname{gtz}(v_{i,d/2-1}) \end{bmatrix}.$$

This truncates all positive values in the tensor to be $\frac{0.5}{i+1}$ at position i, and all nonpositive values to be $-\frac{0.5}{i+1}$. As a result, the next application of layer normalization (with appropriate parameter settings) scales every single value to ± 1, back to Boolean values. In particular, all previously-computed Boolean values are preserved, and the newly-computed dimensions $2k', 2k'+1$ hold the correct Boolean value based on the desired comparison

As a side effect, all previously-computed counts also get changed to ± 1. We will organize the construction so that these values are not used in any further computation. □

4.2.5 Main Result

There may be several ways to perform the simulation of $\mathsf{K}_t[\#, +]$ formulas, but it is convenient to do this by induction over the depth of the formula.

Definition 4.14. The *modal depth* of a formula ϕ or term C, which we notate as $\text{md}(\phi)$, is the maximum level of nesting of # terms. That is,

$$\begin{aligned}
\text{md}(Q_\sigma) &= 0 & \text{md}(1) &= 0 \\
\text{md}(\neg\phi) &= \text{md}(\phi) & \text{md}(\#[\phi]) &= 1 + \text{md}(\phi) \\
\text{md}(\phi_1 \wedge \phi_2) &= \max(\text{md}(\phi_1), \text{md}(\phi_2)) & \text{md}(C_1 + C_2) &= \max(\text{md}(C_1), \text{md}(C_2)) \\
\text{md}(C_1 \leq C_2) &= \max(\text{md}(C_1), \text{md}(C_2))
\end{aligned}$$

Definition 4.15. Fix an alphabet Σ, and assume that the symbol BOS is not in Σ. We say a masked transformer encoder T (as a composition of blocks $T = B_b \circ \cdots \circ B_1 \circ WE$) with d dimensions *simulates* a $\mathsf{K}_t[\#, +]$ formula ϕ if for every input $\mathbf{w} \in \Sigma^*$ with length n and every subformula ψ_k of ϕ,

$$T(\text{BOS} \cdot \mathbf{w})[i+1, 2k : 2k+1] = \begin{cases} \begin{bmatrix} -1 \\ +1 \end{bmatrix} & \text{if } \mathbf{w}, i \vDash \psi_k \\ \\ \begin{bmatrix} +1 \\ -1 \end{bmatrix} & \text{otherwise.} \end{cases}$$

A crucial step in our construction is being able to compose transformers in parallel.

Lemma 4.16. *If T_1 and T_2 are transformers of depth L_1 and L_2 which simulate ϕ_1 and ϕ_2, respectively, then there is a transformer T of depth $L = \max(L_1, L_2)$ which simulates both ϕ_1 and ϕ_2.*

This is straightforward, and is very similar to Lemma 2.6.

Theorem 4.17. *For every $\mathsf{K}_t[\#, +]$ formula ϕ, there exists a masked transformer encoder which simulates ϕ.*

Proof. We induct on the modal depth of ϕ. If ϕ is of modal depth 0, it must be a Boolean combination of Q_σ formulas. This can be simulated in the WE like mentioned in Lemma 2.2

For the inductive step, let ϕ be a $\mathsf{K}_t[\#, +]$ formula of modal depth $m+1$. By Definition 4.14, ϕ is a Boolean combination of:

- Subformulas of modal depth at most m.
- Subformulas of the form $\sum_{k \in K} a_k \#[\psi_k] \geq 0$, where K is a set of indices, a_k are integers, and ψ_k are subformulas of modal depth m.

By the inductive hypothesis, for each subformula ψ_k of modal depth at most m, there is a transformer T_k which simulates it. Parallel-compose all the T_k by Lemma 4.16 into a single transformer. Then we need to perform the following operations in sequence:

1. Compute $\#[\psi_k]$ for all relevant ψ_k, as described in Section 4.2.3.
2. Compute all formulas of the form $\sum_{k \in K} a_k \#[\psi_k] \geq 0$, as described in Section 4.2.4.
3. Compute all Boolean combinations of the above subformulas as necessary.
4. Ensure the BOS position is False.

This can be achieved by adding one block. The first step can be achieved with a self-attention layer. We've described how to compute each of the next three steps individually using a feed-forward layer, but their composition can also be performed with a single feed-forward layer. □

The previous chapter ended with a proof of the depth hierarchy for masked hard attention transformers. This chapter does not! These upper and lower bounds are suspected to not be tight enough, and the logics not well-understood enough, to prove a depth hierarchy. To derive more precise charaterizations of the expressivity of soft attention transformers, and reap the conceptual benefits of these characterizations, poses an interesting challenge for future research.

Bibliography

Raman Arora, Amitabh Basu, Poorya Mianjy, and Anirbit Mukherjee. Understanding deep neural networks with rectified linear units. In *Proceedings of ICLR*, 2018. URL `https://openreview.net/forum?id=B1J_rgWRW`.

Sanjeev Arora and Boaz Barak. *Computational Complexity: A Modern Approach*. Cambridge University Press, 2009.

Jimmy Lei Ba, Jamie Ryan Kiros, and Geoffrey E. Hinton. Layer normalization. In *NIPS 2016 Deep Learning Symposium*, 2016. URL `https://arxiv.org/abs/1607.06450`.

Pablo Barceló, Alexander Kozachinskiy, Anthony Widjaja Lin, and Vladimir Podolskii. Logical languages accepted by transformer encoders with hard attention. In *Proceedings of the Twelfth International Conference on Learning Representations (ICLR)*, 2024. URL `https://openreview.net/forum?id=gbrHZq07mq`.

David A. Barrington, Kevin Compton, Howard Straubing, and Denis Thérien. Regular languages in NC^1. *Journal of Computer and System Sciences*, 44(3):478–499, 1992. ISSN 0022-0000. doi:https://doi.org/10.1016/0022-0000(92)90014-A. URL `https://www.sciencedirect.com/science/article/pii/002200009290014A`.

David A. Mix Barrington, Neil Immerman, and Howard Straubing. On uniformity within NC^1. *Journal of Computer and System Sciences*, 41(3):274–306, 1990. doi:https://doi.org/10.1016/0022-0000(90)90022-D.

David Mix Barrington and Alexis Maciel. Advanced course on computational complexity, 2000. URL `https://people.clarkson.edu/~alexis/PCMI/`. CMI-PCMI Undergraduate Program.

Satwik Bhattamishra, Kabir Ahuja, and Navin Goyal. On the Ability and Limitations of Transformers to Recognize Formal Languages. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 7096–7116, 2020a. doi:10.18653/v1/2020.emnlp-main.576.

Satwik Bhattamishra, Arkil Patel, and Navin Goyal. On the computational power of Transformers and its implications in sequence modeling. In *Proceedings of the 24th Conference on Computational Natural Language Learning (CoNLL)*, pages 455–475, 2020b. doi:10.18653/v1/2020.conll-1.37. URL `https://aclanthology.org/2020.conll-1.37`.

David Chiang and Peter Cholak. Overcoming a theoretical limitation of self-attention. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (ACL)*, pages 7654–7664, May 2022. doi:10.18653/v1/2022.acl-long.527. URL `https://aclanthology.org/2022.acl-long.527`.

David Chiang, Peter Cholak, and Anand Pillay. Tighter bounds on the expressivity of transformer encoders. In *Proceedings of the 40th International Conference on Machine Learning (ICML)*, volume 202 of *Proceedings of Machine Learning Research*, pages 5544–5562, 2023. URL `https://proceedings.mlr.press/v202/chiang23a.html`.

Kousha Etessami and Thomas Wilke. An until hierarchy and other applications of an ehrenfeucht–fraıssé game for temporal logic. *Information and Computation*, 160 (1-2):88–108, 2000. doi:10.1006/inco.1999.2846.

Dov Gabbay, Amir Pnueli, Saharon Shelah, and Jonathan Stavi. On the temporal analysis of fairness. In *Proceedings of the 7th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 163–173, 1980. doi:10.1145/567446.567462.

Michael Hahn. Theoretical limitations of self-attention in neural sequence models. *Transactions of the Association for Computational Linguistics*, 8:156–171, 2020. doi:10.1162/tacl_a_00306. URL `https://aclanthology.org/2020.tacl-1.11`.

Yiding Hao, Dana Angluin, and Robert Frank. Formal language recognition by hard attention Transformers: Perspectives from circuit complexity. *Transactions of the Association for Computational Linguistics*, 10:800–810, 2022. doi:10.1162/tacl_a_00490. URL `https://aclanthology.org/2022.tacl-1.46`.

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition, 2015. URL `http://arxiv.org/abs/1512.03385`. arXiv:1512.03385.

Jeffrey Heinz. The computational nature of phonological generalizations. In Larry Hyman and Frans Plank, editors, *Phonological Typology*, Phonetics and Phonology, chapter 5, pages 126–195. De Gruyter Mouton, 2018.

William Hesse, Eric Allender, and David A. Mix Barrington. Uniform constant-depth threshold circuits for division and iterated multiplication. *Journal of Computer and System Sciences*, 65(4):695–716, 2002. doi:https://doi.org/10.1016/S0022-0000(02)00025-9.

Neil Immerman. *Descriptive Complexity*. Springer, 1999.

Gerhard Jäger and James Rogers. Formal language theory: refining the chomsky hierarchy. *Philosophical Transactions of the Royal Society B: Biological Sciences*, 367(1598):1956–1970, 2012.

Johan Anthony Willem Kamp. *Tense Logic and the Theory of Linear Order*. PhD thesis, University of California, Los Angeles, 1968. URL `https://www.proquest.com/docview/302320357`.

Najoung Kim and Sebastian Schuster. Entity tracking in language models. In Anna Rogers, Jordan Boyd-Graber, and Naoaki Okazaki, editors, *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 3835–3855, 2023. doi:10.18653/v1/2023.acl-long.213. URL `https://aclanthology.org/2023.acl-long.213`.

Klaus-Jörn Lange. Some results on majority quantifiers over words. In *Proceedings of the 19th IEEE Annual Conference on Computational Complexity*, pages 123–129, 2004. doi:10.1109/CCC.2004.1313817.

Robert McNaughton and Seymour A. Papert. *Counter-Free Automata*. MIT Press, 1971. URL https://archive.org/details/CounterFre_00_McNa.

William Merrill and Ashish Sabharwal. A logic for expressing log-precision transformers. In *Advances in Neural Information Processing Systems*, volume 36, pages 52453–52463, 2023a. URL https://proceedings.neurips.cc/paper_files/paper/2023/hash/a48e5877c7bf86a513950ab23b360498-Abstract-Conference.html.

William Merrill and Ashish Sabharwal. The parallelism tradeoff: Limitations of log-precision transformers. *Transactions of the Association for Computational Linguistics*, 11:531–545, 2023b. doi:10.1162/tacl_a_00562. URL https://aclanthology.org/2023.tacl-1.31.

William Merrill and Ashish Sabharwal. The expressive power of transformers with chain of thought. In *Proceedings of the Twelfth International Conference on Learning Representations (ICLR)*, 2024. URL https://openreview.net/pdf?id=NjNGlPh8Wh.

William Merrill, Vivek Ramanujan, Yoav Goldberg, Roy Schwartz, and Noah A. Smith. Effects of parameter norm growth during transformer training: Inductive bias from gradient descent. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 1766–1781, 2021. doi:10.18653/v1/2021.emnlp-main.133. URL https://aclanthology.org/2021.emnlp-main.133.

William Merrill, Ashish Sabharwal, and Noah A. Smith. Saturated transformers are constant-depth threshold circuits. *Transactions of the Association for Computational Linguistics*, 10:843–856, 2022. doi:10.1162/tacl_a_00493. URL https://aclanthology.org/2022.tacl-1.49.

Maxwell Nye, Anders Andreassen, Guy Gur-Ari, Henryk Michalewski, Jacob Austin, David Bieber, David Dohan, Aitor Lewkowycz, Maarten Bosma, David Luan, Charles Sutton, and Augustus Odena. Show your work: Scratchpads for intermediate computation with language models. In *Proceedings of the Workshop on Deep Learning for Code (DL4C)*, 2022. URL https://openreview.net/forum?id=HBlx2idbkbq.

Denis Paperno. On learning interpreted languages with recurrent models. *Computational Linguistics*, 48(2):471–482, June 2022. doi:10.1162/coli_a_00431. URL https://aclanthology.org/2022.cl-2.7.

Doron Peled and Thomas Wilke. Stutter-invariant temporal properties are expressible without the next-time operator. *Information Processing Letters*, 63(5):243–246, 1997. doi:10.1016/S0020-0190(97)00133-6.

Jorge Pérez, Pablo Barceló, and Javier Marinkovic. Attention is Turing-complete. *Journal of Machine Learning Research*, 22:75:1–75:35, 2021. URL http://jmlr.org/papers/v22/20-302.html.

Jean-Éric Pin. How to prove that a language is regular or star-free? In *Language and Automata Theory and Applications (LATA)*, number 12038 in Lecture Notes in Computer Science, pages 68–88, 2020. doi:10.1007/978-3-030-40608-0_5. URL https://hal.science/hal-02949627.

Jorge Pérez, Javier Marinković, and Pablo Barceló. On the Turing completeness of modern neural network architectures. In *Proceedings of the Seventh International Conference on Learning Representations (ICLR)*, 2019. URL https://openreview.net/forum?id=HyGBdo0qFm.

Nicole Schweikardt. Arithmetic, first-order logic, and counting quantifiers. *ACM Transactions on Computational Logic*, 6(3):634–671, jul 2005. doi:10.1145/1071596.1071602.

M. P. Schützenberger. On finite monoids having only trivial subgroups. *Information and Control*, 8(2):190–194, 1965. doi:10.1016/S0019-9958(65)90108-7.

Michael Sipser. *Introduction to the Theory of Computation*. Cengage Learning, 3rd edition, 2013.

Howard Straubing. First-order logic and aperiodic languages: a revisionist history. *ACM SIGLOG News*, 5(3):4–20, jul 2018. doi:10.1145/3242953.3242956. URL https://doi.org/10.1145/3242953.3242956.

Lena Strobl. Average-hard attention transformers are constant-depth uniform threshold circuits, 2023. URL https://arxiv.org/abs/2308.03212. arXiv:2308.03212.

Lena Strobl, Dana Angluin, David Chiang, Jonathan Rawski, and Ashish Sabharwal. Transformers as transducers, 2024a. URL https://arxiv.org/abs/2404.02040. arXiv:2404.02040.

Lena Strobl, William Merrill, Gail Weiss, David Chiang, and Dana Angluin. What formal languages can transformers express? A survey. *Transactions of the Association for Computational Linguistics*, 12:543–561, 2024b. doi:https://doi.org/10.1162/tacl_a_00663. URL https://arxiv.org/abs/2311.00208.

Wolfgang Thomas. Languages, automata, and logic. In Grzegorz Rozenberg and Arto Salomaa, editors, *Handbook of Formal Languages: Volume 3, Beyond Words*, pages 389–455. Springer, 1997. doi:10.1007/978-3-642-59126-6_7.

Johan van Benthem and Thomas Icard. Interleaving logic and counting. *The Bulletin of Symbolic Logic*, 29(4):503–587, 2023. doi:10.1017/bsl.2023.30.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in Neural Information Processing Systems 30 (NeurIPS)*, 2017. URL https://proceedings.neurips.cc/paper/2017/hash/3f5ee243547dee91fbd053c1c4a845aa-Abstract.html.

Qiang Wang, Bei Li, Tong Xiao, Jingbo Zhu, Changliang Li, Derek F. Wong, and Lidia S. Chao. Learning deep Transformer models for machine translation. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics (ACL)*, 2019. doi:10.18653/v1/P19-1176. URL https://aclanthology.org/P19-1176.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed H. Chi, Quoc V. Le, and Denny Zhou. Chain-of-thought prompting elicits reasoning in large language models. In *Advances in Neural Information Processing Systems 35 (NeurIPS)*, pages 24824–24837, 2022. URL https://proceedings.neurips.cc/paper_files/paper/2022/hash/9d5609613524ecf4f15af0f7b31abca4-Abstract-Conference.html.

David H. Wolpert. What is important about the no free lunch theorems? In Panos M. Pardalos, Varvara Rasskazova, and Michael N. Vrahatis, editors, *Black*

Box Optimization, Machine Learning, and No-Free Lunch Theorems, pages 373–388. Springer International Publishing, 2021. ISBN 978-3-030-66515-9. doi:10.1007/978-3-030-66515-9_13. URL https://doi.org/10.1007/978-3-030-66515-9_13.

Andy Yang and David Chiang. Counting like transformers: Compiling temporal counting logic into softmax transformers. In *Proceedings of the Conference on Language Modeling*, 2024. URL https://arxiv.org/abs/2404.04393. To appear.

Andy Yang, David Chiang, and Dana Angluin. Masked hard-attention transformers and Boolean RASP recognize exactly the star-free languages, 2023. URL https://arxiv.org/abs/2310.13897. arXiv:2310.13897.

Shunyu Yao, Binghui Peng, Christos Papadimitriou, and Karthik Narasimhan. Self-attention networks can process bounded hierarchical languages. In *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing (ACL-IJCNLP)*, pages 3770–3785, August 2021. doi:10.18653/v1/2021.acl-long.292. URL https://aclanthology.org/2021.acl-long.292.