Chapter 7

Transformers with softmax attention

In this section, 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*.

7.1 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 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 DLOGTIME-uniform TC^0, which is equivalent to first-order logic with counting quantifiers, addition, and multiplication.

7.1.1 Precision

One key issue is that while unique-hard and average-hard attention 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 7.1. 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 $m \cdot 2^e$.

7.1.2 Arithmetic predicates

We can increase the expressivity of FO by adding more predicates besides $<$.

The syntax of $\mathsf{FO}[+, \times]$ is that of FO, as well as:

$$\begin{aligned} \phi &::= Q_\sigma(t_1) && \sigma \in \Sigma \\ &\mid t_1 = t_2 \mid t_1 < t_2 \\ t &::= x \mid t_1 + t_2 \mid t_1 \times t_2 \end{aligned}$$

We extend the definition of FV and of interpretations I to terms as follows:

$$\begin{aligned} \mathrm{FV}(t_1 + t_2) &= \mathrm{FV}(t_1) \cup \mathrm{FV}(t_2) \\ \mathrm{FV}(t_1 \times t_2) &= \mathrm{FV}(t_1) \cup \mathrm{FV}(t_2) \\ I(t_1 + t_2) &= I(t_1) + I(t_2) \\ I(t_1 \times t_2) &= I(t_1)\, I(t_2). \end{aligned}$$

Then the definition of $\models$ is the same as before. Note that $\forall$ and $\exists$ still quantify over positions of the input string, that is, over $[n]$.

Example 7.2. The following formula of $\mathsf{FO}[+, \times]$ tests whether a number is odd:

$$\mathrm{ODD}(x) = \neg(\exists y.\, y + y = x). \tag{7.1}$$

Exercise 7.3. 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. $\mathrm{DIV}(x, y, q, r)$ such that

$$\mathbf{w}, I \models \mathrm{DIV}(x, y, q, r) \quad \text{if } \lfloor I(x)/I(y) \rfloor = I(q) \text{ and } I(x) \equiv I(r) \bmod I(y)$$

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

Theorem 7.4 (Barrington, Immerman, et al., 1990)**.** $\mathsf{FO}[+, \times]$ *defines exactly the languages in* DLOGTIME*-uniform* AC^0.

Using + and $\times$, we can define many other arithmetic operations.

Theorem 7.5. *The following formulas are definable in* $\mathsf{FO}[+, \times]$*:*

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

(b) $\mathrm{BIT}(x, y, z)$ *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 FO[BIT] predicates ADD and MUL.

Furthermore, while a variable stands for a number in $[n]$, we can also do limited arithmetic on much bigger numbers. A number with n bits can be represented by a formula $\phi(x)$ that is true if the x-th bit is 1. A number with n^k bits can be represented by a formula $\phi(x_0, \ldots, x_{k-1})$ that is true if the $(x_0 + x_1 n + x_2 n^2 + \cdots + x_{k-1} n^{k-1})$-th bit is 1.

Theorem 7.6. *The following operations on* $O(\mathrm{poly}(n))$ *bit integers are expressible in* $\mathsf{FO}[+, \times]$*:*

(a) Addition of two numbers

(b) Comparison of two numbers

(c) Maximum of n *numbers.*

7.1.3 Counting quantifiers

FOC is first order logic with *counting terms* (Benthem and Icard, 2023).

Example 7.7. The majority language,

$$\mathsf{MAJORITY} = \{\mathbf{w} \in \{0, 1\}^* \mid \mathbf{w} \text{ has more 1's than 0's}\}. \tag{7.2}$$

can be defined by the FOC formula

$$\underbrace{(\#z.Q_0(z))}_{\text{number of 0's}} < \underbrace{(\#z.Q_1(z))}_{\text{number of 1's}}. \tag{7.3}$$

Exercise 7.8. Write a FOC formula for

$$\text{PARITY} = \{\mathbf{w} \in \{0,1\}^* \mid \mathbf{w} \text{ has an odd number of 1's}\}. \tag{7.4}$$

The syntax of FOC is:

$$t ::= x \mid \#x.\phi_1 \tag{7.5}$$
$$\phi ::= Q_\sigma(t_1) \qquad \sigma \in \Sigma \tag{7.6}$$
$$\mid t_1 = t_2 \mid t_1 < t_2 \tag{7.7}$$
$$\mid \phi_1 \wedge \phi_2 \mid \phi_1 \vee \phi_2 \mid \neg\phi_1 \tag{7.8}$$
$$\mid \forall x.\phi_1 \mid \exists x.\phi_1 \tag{7.9}$$

We extend the definition of free variables (Eq. (6.5)) with:

$$\text{FV}(x) = \{x\} \tag{7.10}$$
$$\text{FV}(Q_\sigma(t_1)) = \text{FV}(t_1) \qquad \sigma \in \Sigma \tag{7.11}$$
$$\text{FV}(t_1 = t_2) = \text{FV}(t_1) \cup \text{FV}(t_2) \tag{7.12}$$
$$\text{FV}(t_1 < t_2) = \text{FV}(t_1) \cup \text{FV}(t_2) \tag{7.13}$$
$$\text{FV}(\#x.\phi_1) = \text{FV}(\phi_1) \setminus \{x\} \tag{7.14}$$

Terms are interpreted as integers, as follows:

$$x^{\mathbf{w},I} = I(x) \tag{7.15}$$
$$(\#x.\phi_1)^{\mathbf{w},I} = |\{i \in [|\mathbf{w}|] \mid \mathbf{w}, I[x \mapsto i] \models \phi_1\}| \tag{7.16}$$

And we extend the definition of $\models$ (Eq. (6.9)) with:

$$\mathbf{w}, I \models Q_\sigma(t_1) \qquad \text{if } w_{t_1^{\mathbf{w},I}} = \sigma \tag{7.17}$$
$$\mathbf{w}, I \models t_1 = t_2 \qquad \text{if } t_1^{\mathbf{w},I} = t_2^{\mathbf{w},I} \tag{7.18}$$
$$\mathbf{w}, I \models t_1 < t_2 \qquad \text{if } t_1^{\mathbf{w},I} < t_2^{\mathbf{w},I} \tag{7.19}$$

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 7.9 (Barrington, Immerman, et al., 1990)**.** FOM[×] *defines exactly the languages in* DLOGTIME*-uniform* TC^0*.*

In FOM[×] we can do a surprising amount of arithmetic on large numbers.

Theorem 7.10. *The following operations on* $O(\mathrm{poly}(n))$ *bit integers are expressible in* FOM[×]*:*

(a) Iterated addition of n numbers

(b) Multiplication of two numbers

(c) Iterated multiplication of n numbers

(d) Truncated division of two numbers.

Proof. Iterated addition (a) is shown, for example, by Barrington and Maciel (2000, Lecture 7, Section 2), and multiplication (b) is closely related.

Iterated multiplication (c) was proven to be in DLOGTIME-uniform TC^0 by Hesse et al. (2002, Theorem 5.1) and can be used for truncated division (d). □

7.1.4 Main result

Theorem 7.11 (Merrill and Sabharwal, 2023a)**.** *For any* $O(\log n)$*-precision transformer encoder* T *that recognizes a language* L*, there is a formula of* FOM[×] *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 FOM[×].

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 FOM[×] on $O(\log n)$-bit floating-point numbers.

Addition and *multiplication*, already defined on integers in Theorem 7.5, 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:

$$(m_1 \cdot 2^{e_1}) + (m_2 \cdot 2^{e_2}) = \begin{cases} (m_1 + m_2 \cdot 2^{e_2 - e_1}) \cdot 2^{e_1} & e_1 \geq e_2 \\ (m_1 \cdot 2^{e_1 - e_2} + m_2) \cdot 2^{e_2} & e_1 \leq e_2 \end{cases} \tag{7.20}$$

$$(m_1 \cdot 2^{e_1}) \cdot (m_2 \cdot 2^{e_2}) = m_1 m_2 \cdot 2^{e_1 + e_2}. \tag{7.21}$$

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:

$$\sum_{i=0}^{n-1}(m_i \cdot 2^{e_i}) = \left(\sum_{i=0}^{n-1} \underbrace{(m_i \cdot 2^{e_i - e})}_{(*)}\right) \cdot 2^e \tag{7.22}$$

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 (Chiang, 2024)).

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

$$\begin{aligned}
\exp x &= \exp_2(x/\log 2) && (7.23)\\
&= \exp_2(\lfloor x/\log 2 \rfloor)\, \exp_2(x/\log 2 - \lfloor x/\log 2 \rfloor) && (7.24)\\
&= \exp_2(\lfloor x/\log 2 \rfloor)\, \exp(\underbrace{x - \lfloor x/\log 2 \rfloor \log 2}_{r}). && (7.25)
\end{aligned}$$

The first factor is just an integer power of 2. The second factor still involves exp, but now we know that $0 \le r < \log 2$, 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:

$$\exp r = \sum_{i=0}^{\infty} \frac{1}{i!} r^i = \sum_{i=0}^{p-1} \frac{1}{i!} r^i + R_p \tag{7.26}$$

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

$$R_p = \frac{\exp z}{p!} r^p < \frac{\exp r}{p!} r^p < \frac{2}{p!} r^p \le \frac{2}{2^p} r^p < \frac{1}{2^{p-1}}. \tag{7.27}$$

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. (7.26) 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). □

7.2 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. (2020) 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 $\mathsf{K}_t[\#, +]$.

7.2.1 $\mathsf{K}_t[\#, +]$

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

$$t ::= \#[\phi_1] \tag{7.28}$$
$$\mid t_1 + t_2 \tag{7.29}$$
$$\phi ::= Q_\sigma \qquad \sigma \in \Sigma \tag{7.30}$$
$$\mid \phi_1 \wedge \phi_2 \mid \neg\phi_1 \tag{7.31}$$
$$\mid t_1 = t_2 \mid t_1 < t_2 \tag{7.32}$$

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.

$$\#[\phi_1]^{\mathbf{w},i} = |\{j \leq i \mid \mathbf{w}, j \models \phi_1\}| \tag{7.33}$$
$$(t_1 + t_2)^{\mathbf{w},i} = t_1^{\mathbf{w},i} + t_2^{\mathbf{w},i} \tag{7.34}$$

And we define the semantics as follows:

$$\mathbf{w}, i \models Q_\sigma \qquad \text{iff } \mathbf{w}[i] = \sigma \tag{7.35}$$
$$\mathbf{w}, i \models \phi_1 \wedge \phi_2 \qquad \text{iff } \mathbf{w}, i \models \phi_1 \text{ and } \mathbf{w}, i \models \phi_2 \tag{7.36}$$
$$\mathbf{w}, i \models \neg\phi \qquad \text{iff } \mathbf{w}, i \not\models \phi \tag{7.37}$$
$$\mathbf{w}, i \models t_1 = t_2 \qquad \text{iff } t_1^{\mathbf{w},i} = t_2^{\mathbf{w},i} \tag{7.38}$$
$$\mathbf{w}, i \models t_1 < t_2 \qquad \text{iff } t_1^{\mathbf{w},i} < t_2^{\mathbf{w},i} \tag{7.39}$$

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

Example 7.12. Below are some example $\mathsf{K}_t[\#, +]$ formulas and the languages they define:

Language	Formula
$\mathsf{a}^*\mathsf{b}^*$	$\#[Q_\mathsf{a} \wedge (\#[Q_\mathsf{b}] \geq 1)] = 0$
$\mathsf{a}^*\mathsf{b}^*\mathsf{a}^*$	$\#[Q_\mathsf{b} \wedge \#[Q_\mathsf{a} \wedge (\#[Q_\mathsf{b}] \geq 1)] \geq 1] = 0$
$\mathsf{a}^n\mathsf{b}^n\mathsf{c}^n$	$\#[Q_\mathsf{b} \wedge (\#[Q_\mathsf{c}] = 0)] = \#[Q_\mathsf{b}]$ $\wedge\, \#[Q_\mathsf{a} \wedge (\#[Q_\mathsf{b} \vee Q_\mathsf{c}] = 0)] = \#[Q_\mathsf{a}]$ $\wedge\, \#[Q_\mathsf{a}] = \#[Q_\mathsf{b}] \wedge \#[Q_\mathsf{b}] = \#[Q_\mathsf{c}] \wedge \#[Q_\mathsf{c}] = \#[Q_\mathsf{a}]$
Dyck-1	$\big(\#[Q_(] = \#[Q_)]\big) \wedge \big(\#[\#[Q_)] > \#[Q_(]] = 0\big)$
`hello`	$\#[\top] = 5 \wedge Q_\mathsf{o} \wedge \#[Q_1 \wedge \#[Q_\mathsf{e} \wedge \#[Q_\mathsf{h}] = 1] = 1] = 2$

7.2.2 Boolean and count representations

Many proofs of transformer lower bounds ignore the effects of layer normalization (Section 5.1.3). 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:

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

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. (5.19) and (5.20)). 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 $\mathbb{I}[\phi_k(i)]$:

$$\begin{aligned} \mathbf{A}[0, 2k : 2k+1] &= \begin{bmatrix} 1 \\ -1 \end{bmatrix} \\ \mathbf{A}[i, 2k : 2k+1] &= \begin{bmatrix} -2\phi_k(i) + 1 \\ 2\phi_k(i) - 1 \end{bmatrix} \qquad i > 0. \end{aligned} \tag{7.40}$$

Similarly, each count term C_k is stored as:

$$\begin{aligned} \mathbf{A}[0, 2k : 2k+1] &= \begin{bmatrix} 0 \\ 0 \end{bmatrix} \\ \mathbf{A}[1, 2k : 2k+1] &= \begin{bmatrix} -\frac{C_k(i)}{i+1} \\ \frac{C_k(i)}{i+1} \end{bmatrix} \qquad i > 0. \end{aligned} \tag{7.41}$$

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.

7.2.3 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 7.13. *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

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

□

Actually, remember that we are using layer normalization, so this vector might actually be scaled by some factor. But this won't affect the correctness of the construction.

7.2.4 Linear constraints

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

$$\sum_{k \in K} a_k C_k(i) \geq 0 \tag{7.42}$$

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 7.14. *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. (7.41)). Let a_k for $k \in K$ be integer coefficients as in Eq. (7.42). 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. (7.42) is true or false (using the representation in Eq. (7.40)).*

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.

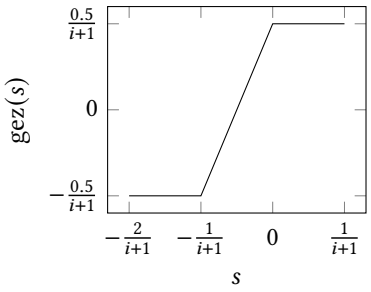
Second, we use a FFNN to compute, at each position i, the linear combination

$$S(i) = \frac{\sum_{k \in K} a_k C_k(i)}{i+1} = \sum_{k \in K} a_k \frac{C_k(i)}{i+1}. \tag{7.43}$$

To test whether this is nonnegative, we construct a feed-forward layer that computes the function

$$\mathrm{gez}(s) = \max\left(-\frac{0.5}{i+1}, \min\left(\frac{0.5}{i+1}, s\right)\right) \tag{7.44}$$

$$= \mathrm{ReLU}\left(s + \frac{1}{i+1}\right) - \mathrm{ReLU}\,(s) - \frac{0.5}{i+1} \tag{7.45}$$



(This is where $\frac{1}{i+1}$ gets used.)

Observe that $\mathrm{gez}(S(i))$ equals $\frac{0.5}{i+1}$ if $\sum_k a_k C_k(i) \geq 0$, and $-\frac{0.5}{i+1}$ otherwise. This is because the counts must be integers, so if $\sum_k a_k C_k(i) < 0$, then $\sum_k a_k C_k(i) \leq -1$.

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

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

This truncates all positive values 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. □

7.2.5 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 7.15. The *modal depth* of a formula ϕ or term C, which we notate as $\mathrm{md}(\phi)$, is the maximum level of nesting of # terms. That is,

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

Definition 7.16. 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 $T =$

$B_b \circ \cdots \circ B_1 \circ \mathrm{WE}$) 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,

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

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

Lemma 7.17. *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 6.20.

Theorem 7.18. *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 as mentioned in Lemma 6.15.

For the inductive step, let ϕ be a $\mathsf{K}_t[\#, +]$ formula of modal depth $m+1$. By Definition 7.15, 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 7.17 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 7.2.3.
2. Compute all formulas of the form $\sum_{k \in K} a_k \#[\psi_k] \geq 0$, as described in Section 7.2.4.
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 characterizations of the expressivity of soft attention transformers, and reap the conceptual benefits of these characterizations, poses an interesting challenge for future research.

Exercise 7.19. Let $\mathsf{K}_t[\#, +, \mathrm{ODD}]$ be the same as $\mathsf{K}_t[\#, +]$, but extended with a built-in predicate ODD such that $\mathbf{w}, i \models \mathrm{ODD}$ iff i is an odd number.

(a) Write a formula of $\mathsf{K}_t[\#, +, \mathrm{ODD}]$ for the language $(\mathtt{01})^*$.

(b) How would you extend the construction of Theorem 7.18 to handle formulas of $\mathsf{K}_t[\#, +, \mathrm{ODD}]$?

Exercise 7.20. Consider a variant of $\mathsf{K}_t[\#, +]$ called $\mathsf{K}_t[\%, +]$, which, instead of counting terms $\#[\phi_1]$, has averaging terms $\%[\phi_1]$, which are interpreted as rational numbers.

$$\%[\phi_1]^{\mathbf{w},i} = \frac{|\{j \leq i \mid \mathbf{w}, j \models \phi_1\}|}{i+1}$$

(a) Explain why $\mathsf{K}_t[\%, +]$ cannot define the language $\{\mathtt{a}\}$.

(b) How would you modify the construction of Theorem 7.18 to handle formulas of $\mathsf{K}_t[\%, +]$?