Chapter 5

Transformers

5.1 Model

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. (2024).

Transformers are composed of an input layer (Section 5.1.1), one or more hidden layers (Section 5.1.4), and an output layer. The inputs and outputs of the layers are sequences of vectors, which we treat as members of $(\mathbb{R}^d)^*$.

5.1.1 Input layer

Strings are initially mapped to sequences of vectors by $emb\colon \Sigma^* \xrightarrow{\text{lp}} (\mathbb{R}^d)^*$, 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 $n \in \mathbb{N}_{>0}$:

$$emb(w_0 \cdots w_{n-1})[i] = \text{WE}(w_i) + \text{PE}_n(i). \tag{5.1}$$

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

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

$$\text{PE}_n(i)[j] = \begin{cases} \sin(10000^{-j/d} \cdot i) & \text{if } j \text{ even} \\ \cos(10000^{-(j-1)/d} \cdot i) & \text{if } j \text{ odd.} \end{cases} \tag{5.2}$$

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

5.1.2 Attention

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

$$
\begin{aligned}
att\colon (\mathbb{R}^d)^* &\xrightarrow{\text{lp}} (\mathbb{R}^d)^* \\
\mathbf{X} &\mapsto \mathbf{Y} && (5.3)\\
\mathbf{Q}[i] &= \mathbf{W}^{\mathrm{Q}}(\mathbf{X}[i]) && (5.4)\\
\mathbf{K}[j] &= \mathbf{W}^{\mathrm{K}}(\mathbf{X}[j]) && (5.5)\\
\mathbf{V}[j] &= \mathbf{W}^{\mathrm{V}}(\mathbf{X}[j]) && (5.6)\\
\mathbf{s}[i,j] &= \frac{\mathbf{Q}[i]\cdot\mathbf{K}[j]}{\sqrt{d_{\text{hid}}}} && (5.7)\\
\alpha[i,:] &= \operatorname{softmax}(\mathbf{s}[i,:]) && (5.8)\\
&= \frac{\exp \mathbf{s}[i,:]}{\displaystyle\sum_{j\in[d]} \exp \mathbf{s}[i,j]} && (5.9)\\
\mathbf{Y}[i] &= \sum_{j\in[n]} \alpha[i,j]\,\mathbf{V}[j] && (5.10)
\end{aligned}
$$

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, but we don't use it because it can be emulated using single-head self-attention.

Attention masking In *future-masked* (also known as *causally*-masked) self attention, a term $m(i,j)$ is added to Eq. (5.7) to force every position to attend only to preceding positions:

$$
m(i,j) = \begin{cases} 0 & \text{if } j \le i \\ -\infty & \text{otherwise.} \end{cases} \tag{5.11}
$$

(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 ($j > i$).

Hard attention Some theoretical analyses simplify attention by replacing the softmax with variants that focus attention only on the position(s) with the maximum value, breaking ties in various ways.

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. (5.8) with:

$$\alpha[i, j] = \mathbb{I}[j = \min M(\mathbf{s}[i, :])]. \tag{5.12}$$

whereas in *average*-hard attention, the maximal elements share weight equally:

$$\alpha[i, j] = \frac{\mathbb{I}[j \in M(\mathbf{s}[i, :])]}{|M(\mathbf{s}[i, :])|}. \tag{5.13}$$

Leftmost-hard attention was previously called *hard* attention by Hahn (2020) and *unique-hard* attention by Hao et al. (2022). One may also consider rightmost-hard attention, in which the rightmost maximal element is used. Average-hard attention was also called *hard* attention by Pérez, Barceló, et al. (2021) and *saturated* attention by Merrill, Sabharwal, et al. (2022), and has been argued to be a realistic approximation to how trained transformers behave in practice (Merrill, Ramanujan, et al., 2021). Neither type of hard attention should be confused with the concept of hard attention used in computer vision (e.g., Xu et al., 2015).

5.1.3 Layer normalization

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

$$\begin{aligned} \mathit{norm} \colon \mathbb{R}^d &\to \mathbb{R}^d \\ \mathbf{x} &\mapsto \gamma \odot \frac{\mathbf{x} - \bar{\mathbf{x}}}{\sqrt{\mathrm{Var}(\mathbf{x}) + \varepsilon}} + \beta \end{aligned} \tag{5.14}$$

where $\odot$ is component-wise multiplication,

$$\bar{\mathbf{x}} = \frac{1}{d} \sum_{i \in [d]} \mathbf{x}_i \tag{5.15}$$

$$\mathrm{Var}(\mathbf{x}) = \frac{1}{d} \sum_{i \in [d]} (\mathbf{x}_i - \bar{\mathbf{x}})^2 \tag{5.16}$$

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, Barceló, et al., 2021).

5.1.4 Hidden layers

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

$$\begin{aligned} \mathit{layer}\colon (\mathbb{R}^d)^* &\xrightarrow{\text{lp}} (\mathbb{R}^d)^* \\ \mathbf{X} &\mapsto \mathbf{Y} \text{ where} \\ \mathbf{H} &= \mathit{norm}_1(\mathbf{X} + \mathit{att}(\mathbf{X})) \\ \mathbf{Y} &= \mathit{norm}_2(\mathbf{H} + \mathit{ffn}(\mathbf{H})) \end{aligned} \tag{5.17}$$

and the *pre-norm* variant (Wang et al., 2019) has

$$\begin{aligned} \mathbf{H} &= \mathbf{X} + \mathit{att}(\mathit{norm}_1(\mathbf{X})) \\ \mathbf{Y} &= \mathbf{H} + \mathit{ffn}(\mathit{norm}_2(\mathbf{H})) \end{aligned} \tag{5.18}$$

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. In applications, the hidden layer usually has $4d$ dimensions, but theoretical constructions use as many or as few dimensions as needed.
- 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*.

5.1.5 Transformer encoders

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

$$\begin{aligned}
\mathit{tfr} \colon \Sigma^* &\xrightarrow{\text{lp}} (\mathbb{R}^d)^* \\
\mathbf{w} &\mapsto \mathbf{A}^{(L)} \text{ where} \\
\mathbf{A}^{(0)} &= \mathit{emb}(w) \\
\mathbf{A}^{(1)} &= \mathit{layer}_1(\mathbf{A}^{(0)}) \\
&\vdots \\
\mathbf{A}^{(L)} &= \mathit{layer}_L(\mathbf{A}^{(L-1)})
\end{aligned} \tag{5.19}$$

where

- *emb* is an input layer
- L is called the *depth*
- each layer_ℓ for $\ell \in [L]$ is a post-norm transformer layer (5.17).

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

$$\begin{aligned}
\mathit{tfr} \colon \Sigma^* &\xrightarrow{\text{lp}} (\mathbb{R}^d)^* \\
\mathbf{w} &\mapsto \mathit{norm}(\mathbf{A}^{(L)}) \text{ where} \\
\mathbf{A}^{(0)} &= \mathit{emb}(\mathbf{w}) \\
\mathbf{A}^{(1)} &= \mathit{layer}_1(\mathbf{A}^{(0)}) \\
&\vdots \\
\mathbf{A}^{(L)} &= \mathit{layer}_L(\mathbf{A}^{(L-1)})
\end{aligned} \tag{5.20}$$

where

- *emb* is an input layer
- L is called the *depth*
- each layer_ℓ for $\ell \in 1, \ldots, L$ is a pre-norm transformer layer (5.18)
- *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

$$\begin{aligned} \mathit{out} \colon \mathbb{R}^d &\to \mathbb{R} \\ \mathbf{h} &\mapsto \mathbf{W}\mathbf{h} + b \qquad\qquad i \in [n] \end{aligned} \tag{5.21}$$

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

Exercise 5.1. One very common operation in a transformer is for each position to look at the position to its immediate left. (Let's say the leftmost position looks at itself.) Assume we have a sequence of vectors

$$\mathbf{X} = \begin{bmatrix} x_0 \\ 1 \\ 0 \\ 0 \\ 0 \end{bmatrix}, \begin{bmatrix} x_1 \\ 1 \\ 1 \\ 1 \\ 0 \end{bmatrix}, \begin{bmatrix} x_2 \\ 1 \\ 2 \\ 4 \\ 0 \end{bmatrix}, \ldots, \begin{bmatrix} x_i \\ 1 \\ i \\ i^2 \\ 0 \end{bmatrix}, \ldots, \begin{bmatrix} x_{n-1} \\ 1 \\ n-1 \\ (n-1)^2 \\ 0 \end{bmatrix},$$

for some sequence $x_0, \ldots, x_{n-1} \in \mathbb{R}$. Write down an *average*-hard attention transformer layer f that computes

$$f(\mathbf{X}) = \begin{bmatrix} x_0 \\ 1 \\ 0 \\ 0 \\ x_0 \end{bmatrix}, \begin{bmatrix} x_1 \\ 1 \\ 1 \\ 1 \\ x_0 \end{bmatrix}, \begin{bmatrix} x_2 \\ 1 \\ 2 \\ 4 \\ x_1 \end{bmatrix}, \ldots, \begin{bmatrix} x_i \\ 1 \\ i \\ i^2 \\ x_{i-1} \end{bmatrix}, \ldots, \begin{bmatrix} x_{n-1} \\ 1 \\ n-1 \\ (n-1)^2 \\ x_{n-2} \end{bmatrix}.$$

Hint: The function $2j - j^2$ is maximized when $j = 1$. Can you tweak this function to be maximized at $j = i - 1$, and can you construct the f so that $\mathbf{s}[i, j]$ equals this function?

5.2 Expressivity

The two seminal papers in the area of expressivity of transformers are often summarized as follows:

> Transformers are Turing-complete (Pérez, Marinković, et al., 2019; Pérez, Barceló, 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).

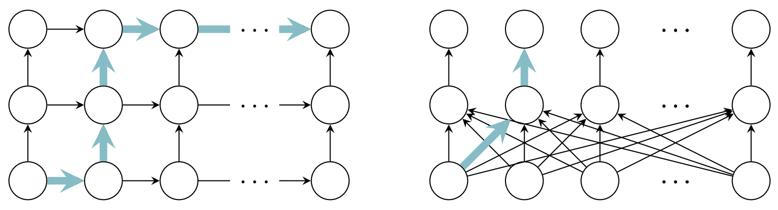


Figure 5.1: Thought of as computation graphs, an RNN (a) has a path of length $O(n)$, but a transformer (b) only has paths of length $O(1)$.

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 our goals in the next three chapters is to disentangle these assumptions and help you to understand how all the results in this area of research fit together.

5.2.1 Parallelism

In *Theory of Computing*, you learned about the Chomsky hierarchy of regular languages, context-free languages, context-sensitive languages, and Turing-recognizable (or recursively enumerable) languages, and in the previous chapter, we saw how RNNs could be related to two levels of that hierarchy (regular and Turing-recognizable).

With transformers, we will see that under the right assumptions, they, too, can be related to Turing machines. However, none of the other levels of the Chomsky hierarchy seem to be a good fit for transformers. In particular, there are non-regular languages that transformers can express and some regular languages that transformers apparently can't express.

In a word, the reason is *parallelism*. If you think of a neural network as a computation graph, then an RNN has a path of length n (Fig. 5.1a). So does a finite automaton. But a transformer only has paths of bounded length (Fig. 5.1b).

One reason for the difference is trainability: long paths can lead to vanishing or exploding gradients. Another reason, and the reason given in the original transformer paper (Vaswani et al., 2017), is that networks with short paths are more parallelizable on GPUs.

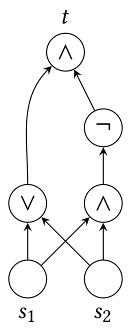
5.2.2 Circuit complexity

In complexity theory, one way of modeling parallelizability is *circuit complexity*. This section, which is an expanded version of Section 5.2 of the survey by Strobl et al. (2024), gives a brief introduction to circuit complexity. For a more detailed treatment, please see the textbook by Arora and Barak (2009).

In this section, 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 $[n]$.

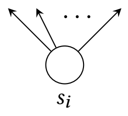
Circuits operate on binary values, so for the rest of this section, we assume $\Sigma = \{0, 1\}$. (If we want to use circuits to model sets of strings over an alphabet Σ, we can choose a fixed-length encoding of the symbols of Σ as strings of $b = \log |\Sigma|$ bits and encode the value of the i-th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

Example 5.2. 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.

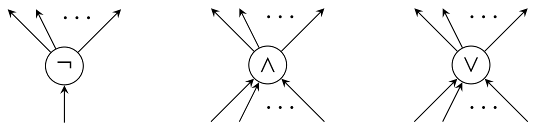


Definition 5.3 (Boolean circuits)**.** A *(Boolean) circuit* C with input length n is a directed acyclic graph with

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



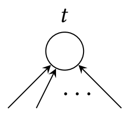
2. zero or more fan-in (in-degree) *gate* nodes, each labeled with a function:



- 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.)

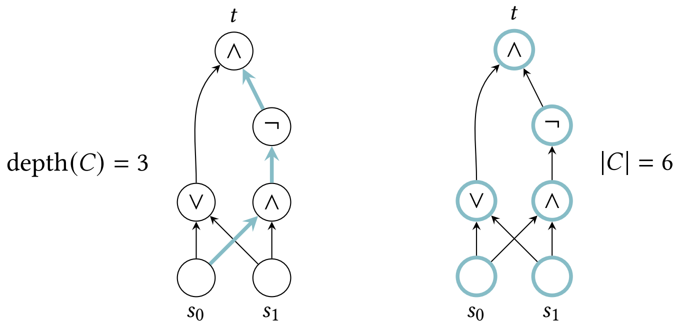
3. A node t, which can be either an input or gate node, is designated the *output* of the circuit.



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. Thus, we can think of the circuit as computing a function $C\colon \{0,1\}^* \to \{0,1\}$, mapping each input string $\mathbf{w}$ to the value of t.

The *depth* of C, denoted $\operatorname{depth}(C)$, 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 5.4. The longest path in C in Example 5.2 is 3, so $\operatorname{depth}(C) = 3$. The number of nodes in C is 6, so $|C| = 6$.



Definition 5.5 (Boolean circuit families)**.** A *circuit family* is a sequence $C = (C_n)_{n\in\mathbb{N}}$ such that for each n, C_n is a circuit with input length n. We treat C as a function on $\{0,1\}^*$ as follows.

For every $\mathbf{w} \in \{0,1\}^*$ with length n, $C(\mathbf{w}) = C_n(\mathbf{w})$. Then the language defined by C is $L(C) = \{\mathbf{w} \in \{0,1\}^* \mid C(\mathbf{w}) = 1\}$. The *depth* and *size* of C are the functions $n \mapsto \operatorname{depth}(C_n)$ 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 5.6 (AC^k, TC^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.
- TC^k is like AC^k, but also allows MAJORITY gates, which have unbounded fan-in and output 1 iff at least half of their inputs are 1.
- 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 5.7. 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 *uniformity* restriction, which says that, given n, the circuit C_n must be constructible under some limitation on computational resources, in the following sense.

Definition 5.8 ($\mathsf{DLOGTIME}$ uniformity, Barrington et al., 1990)**.** Let $C = (C_n)_{n\in\mathbb{N}}$ be a circuit family, and assume that the nodes of C_n are numbered from 0 to $|C_n| - 1$. We say that C is $\mathsf{DLOGTIME}$-*uniform* if there is a (deterministic) Turing machine that runs in logarithmic time and accepts those tuples $\langle f, i, j, 1^n\rangle$ such that in C_n, node i has label f and there is an edge from node i to node j.

For the rest of this section, whenever we mention a circuit complexity class, we mean the $\mathsf{DLOGTIME}$-uniform version of it.

Figure 5.2 shows the language classes in the Chomsky hierarchy and the circuit classes we'll use in this course. AC^0 is the smallest circuit complexity class we'll be interested in. As a reminder, the 0 means that the circuit families in this class have bounded ($O(1)$) depth.

The classic examples of languages *not* in this class are (Furst et al., 1984):

$$\begin{aligned}\mathsf{PARITY} &= \{\mathbf{w} \in \{\mathtt{0}, \mathtt{1}\}^* \mid \mathbf{w} \text{ has an odd number of } \mathtt{1}\text{'s}\}\\ \mathsf{MAJORITY} &= \{\mathbf{w} \in \{\mathtt{0}, \mathtt{1}\}^* \mid \mathbf{w} \text{ has more } \mathtt{1}\text{'s than } \mathtt{0}\text{'s}\}.\end{aligned}$$

TC^0 contains both of the above languages, and many more. What are some languages not in TC^0? This question is especially interesting, because, as we will see (Section 8.1), we think such languages would be too difficult for transformers. There is an open question of whether $\mathsf{TC}^0 = \mathsf{NC}^1$ (analogous to the more famous question of whether $\mathsf{P} = \mathsf{NP}$), and it's widely believed that $\mathsf{TC}^0 \neq \mathsf{NC}^1$. If they are in fact different, then any NC^1-complete language is not in TC^0.

- 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.

- 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 $\pi\colon [k] \rightarrow [k]$, 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_5, the permutation (01) swaps 0 and 1, while the permutation (01234) cycles 0 to 1, 1 to 2, and so on, and 4 to 0. So $(01)(01) \in \mathsf{W}(S_5)$ but $(01234)(01234) \notin \mathsf{W}(S_5)$. These languages are easy for finite automata to recognize, but difficult with only fixed computation depth.

5.2.3 Overview of results

Where do transformers sit in Fig. 5.2?

- If attention is simplified to *unique-hard* attention, in which attention is entirely focused on one position, then transformers are in AC^0 (Chapter 7) and therefore cannot recognize PARITY.

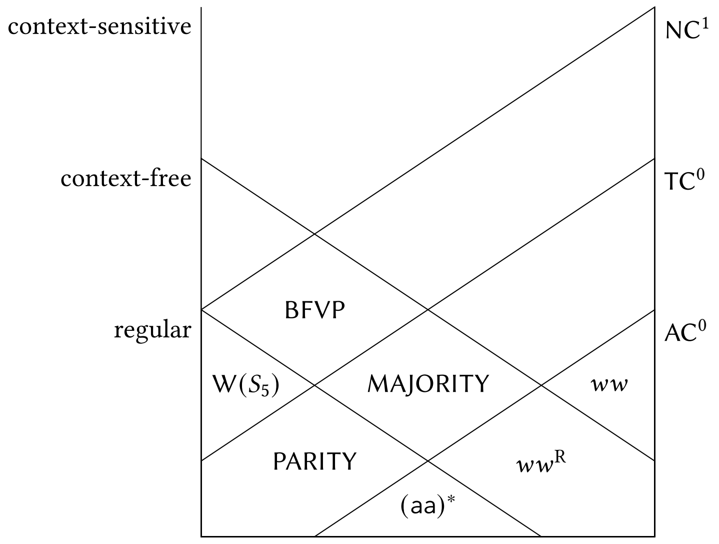


Figure 5.2: Some complexity classes defined by circuit families and logics, compared with the perhaps more familiar Chomsky hierarchy. Assumes $\mathsf{TC}^0 \neq \mathsf{NC}^1$ and $\mathsf{L} \neq \mathsf{NL}$. Circuit classes are DLOGTIME-uniform.

- Otherwise, transformers appear to be in TC^0 (Section 8.1).
- But if we allow transformers to take *intermediate steps*, they are Turing-complete (Chapter 9).

Bibliography

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

Ba, Jimmy Lei, Jamie Ryan Kiros, and Geoffrey E. Hinton (2016). Layer normalization. In: *NIPS 2016 Deep Learning Symposium*.

Barrington, David A. Mix, Neil Immerman, and Howard Straubing (1990). On uniformity within NC^1. In: *Journal of Computer and System Sciences* 41.3, pp. 274–306.

Chiang, David and Peter Cholak (May 2022). Overcoming a theoretical limitation of self-attention. In: *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (ACL)*, pp. 7654–7664.

Furst, Merrick, James B. Saxe, and Michael Sipser (1984). Parity, circuits, and the polynomial-time hierarchy. In: *Mathematical Systems Theory* 17, pp. 13–27.

Hahn, Michael (2020). Theoretical limitations of self-attention in neural sequence models. In: *Transactions of the Association for Computational Linguistics* 8, pp. 156–171.

Hao, Yiding, Dana Angluin, and Robert Frank (2022). Formal language recognition by hard attention Transformers: perspectives from circuit complexity. In: *Transactions of the Association for Computational Linguistics* 10, pp. 800–810.

He, Kaiming, Xiangyu Zhang, Shaoqing Ren, and Jian Sun (2015). Deep residual learning for image recognition. arXiv:1512.03385.

Merrill, William, Vivek Ramanujan, Yoav Goldberg, Roy Schwartz, and Noah A. Smith (2021). 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)*, pp. 1766–1781.

Merrill, William, Ashish Sabharwal, and Noah A. Smith (2022). Saturated transformers are constant-depth threshold circuits. In: *Transactions of the Association for Computational Linguistics* 10, pp. 843–856.

Pérez, Jorge, Pablo Barceló, and Javier Marinkovic (2021). Attention is Turing-complete. In: *Journal of Machine Learning Research* 22, 75:1–75:35.

Pérez, Jorge, Javier Marinković, and Pablo Barceló (2019). On the Turing completeness of modern neural network architectures. In: *Proceedings of the Seventh International Conference on Learning Representations (ICLR)*.

Strobl, Lena, William Merrill, Gail Weiss, David Chiang, and Dana Angluin (2024). What formal languages can transformers express? A survey. In: *Transactions of the Association for Computational Linguistics*. To appear.

Vaswani, Ashish, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin (2017). Attention is all you need. In: *Advances in Neural Information Processing Systems 30 (NeurIPS)*.

Wang, Qiang, Bei Li, Tong Xiao, Jingbo Zhu, Changliang Li, Derek F. Wong, and Lidia S. Chao (2019). Learning deep Transformer models for machine translation. In: *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics (ACL)*.

Xu, Kelvin, Jimmy Ba, Ryan Kiros, Kyunghyun Cho, Aaron Courville, Ruslan Salakhudinov, Rich Zemel, and Yoshua Bengio (2015). Show, attend and tell: neural image caption generation with visual attention. In: *Proceedings of the 32nd International Conference on Machine Learning*, pp. 2048–2057.

Yao, Shunyu, Binghui Peng, Christos Papadimitriou, and Karthik Narasimhan (Aug. 2021). 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)*, pp. 3770–3785.