THE IMPACT OF RANDOMNESS IN COT: ADDRESSING STOCHASTICITY IN FORMAL MODELS

Anonymous authors
Paper under double-blind review

ABSTRACT

Chain-of-Thought (CoT) has proved to be a highly effective method to improve the accuracy of Large Language Models (LLMs). Although there are intuitive explanations for its effectiveness and ongoing efforts to formalize its power, its limits remain unclear. One approach to understanding CoT is grounded on computational complexity theory and relies on modeling LLMs with CoT as a family of shallow circuits equipped with a mechanism to do recursion. Such formalizations yield upper bounds on the power of bounded CoT, as well as lower bounds on the required number of steps necessary to solve particular problems such as PARITY or REACHABILITY. However, many of these abstractions ignore the *stochastic* component of CoT: in practice, LLMs can sample from a probability distribution over the next token at each step, rather than selecting it deterministically, as assumed by previous formal models. In this paper we address this limitation by introducing a more realistic model that includes the stochastic components of LLMs. We show that the complexity classes associated to this model are robust and well-behaved. Moreover, we adapt existing theoretical results on upper and lower bounds for *deterministic* models of LLMs to our setting. Finally, we show how *deterministic* and *stochastic* models relate to one another and with Turing Machines. These results suggest a framework for assessing when a stochastic abstraction of LLMs is necessary and when a deterministic abstraction is sufficiently faithful for the question under study. By extending previous theoretical findings to the stochastic setting, our work provides a stronger theoretical foundation for the experiments reported in prior work.

1 INTRODUCTION

Since the introduction of the transformer architecture (Vaswani et al., 2017), transformers have become central to language modeling (Dai et al., 2019; Brown et al., 2020) and have achieved strong results in other domains (Jumper et al., 2021; Lin et al., 2023). Nowadays, they underpin modern large language models (LLMs) (Brown et al., 2020), which exhibit strong capabilities in challenging tasks such as mathematical problem-solving and code generation (Chowdhery et al., 2023; Chen et al., 2021). A widely used way to enhance these capabilities is to allow models to generate intermediate reasoning steps before producing their final answers, a procedure known as *chain of thought* (CoT) (Wei et al., 2022). Empirical studies have shown that CoT can substantially improve reasoning performance (Wei et al., 2022; Kojima et al., 2022), yet its computational limits are not well understood. Moreover, CoT has emerged as a promising tool for monitoring and controlling AI systems, as inspecting intermediate reasoning can help detect and prevent undesirable behavior (Baker et al., 2025; Korbak et al., 2025). However, this opportunity is fragile: models can produce misleading or incomplete explanations when CoT is not necessary for the behavior being monitored (Emmons et al., 2025), and optimization against a monitor can encourage models to obscure their reasoning (Baker et al., 2025). Therefore, a theory characterizing the computational power and limitations of CoT is particularly relevant to understanding the scope and limitations of AI control techniques that rely on monitoring intermediate reasoning.

A central theoretical question, then, is how much computational power this additional resource provides, and how that power scales with the number of CoT steps. One approach to this question is founded on the theory of circuit complexity: under suitable assumptions on depth and numerical precision, a transformer's forward pass can be simulated by a shallow circuit, so auto-regressive

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

generation can be viewed as an iteration of such circuits (Merrill & Sabharwal, 2023; Li et al., 2024). This perspective makes it possible to derive both upper bounds on the computational power of transformers and lower bounds on the number of CoT steps required for particular tasks, drawing on established results in circuit complexity. For instance, Li et al. (2024) show that, in their constant-depth, constant-precision model, transformers with bounded embedding dimension and $O(\log n)$ CoT steps[1] can only compute functions in AC^0; and therefore they cannot solve PARITY. Related work has also shown how transformers with CoT can evaluate arithmetic expressions and implement dynamic programming algorithms (Feng et al., 2023), and has established lower bounds on the number of CoT steps required for tasks such as PARITY and REACHABILITY in the unique hard-attention setting (Amiri Bavandpour et al., 2025).

These characterizations depend on the precise formalization of the transformer architecture. Relevant choices include whether the attention mechanism selects the value at a single position maximizing the attention score, averages the values over all maximizing positions, or computes a weighted average using softmax weights (Hao et al., 2022; Chiang, 2025); whether numerical precision is constant or depends on the input length (Merrill & Sabharwal, 2023; Li et al., 2024; Chiang, 2025); and how the embedding dimension scales with the input length (Li et al., 2024). The precise formulation of layer normalization also matters, and in many cases it is omitted for simplicity (Chiang & Cholak, 2022; Brody et al., 2023; Merrill & Sabharwal, 2024).

Many of these works make another simplifying assumption: next-token selection is deterministic, typically implemented by taking an argmax over the probability distribution at the end of each forward pass. This raises the question of whether their conclusions remain valid when tokens are sampled from the output distribution, as is common in practice. In particular, upper and lower bounds in the context of bounded and deterministic CoT do not automatically extend to the stochastic setting. In this work, we develop a complexity-theoretic framework for probabilistic transformers with CoT under the constant-depth, constant-precision assumptions of Li et al. (2024). At each auto-regressive step, the next token is sampled from the output distribution, and we introduce bounded-error complexity classes $\mathsf{BPCoT}[T(n), d(n)]$ parameterized by CoT length and embedding dimension. We establish basic robustness properties of these classes, including closure under Boolean operations and error reduction, and extend several central results on deterministic CoT to this setting: the logarithmic-CoT upper bound and circuit-simulation construction of Li et al. (2024), the circuit-depth bounds of Xu & Sato (2026), and the sensitivity-based lower bounds of Chen et al. (2026). Finally, we relate these models to classical randomized computation: with polynomially bounded resources, nonuniform probabilistic CoT captures $\mathsf{BPP/poly} = \mathsf{P/poly}$, whereas its logspace-uniform counterpart captures BPP. Together, these theorems clarify which existing results remain valid in the stochastic setting, and highlight the relation between CoT, randomness and non-uniformity.

Related work We observe that probabilistic CoT has already received some theoretical attention. Nowak et al. (2024) formalize CoT in terms of distributions over output strings and show that suitable CoT-augmented transformers can represent the same distributions as probabilistic Turing machines when allowing unbounded CoT. Mirtaheri et al. (2025) approximate constant-length sampled CoT using randomized TC^0 circuits and show, under some common complexity-theoretical assumption, that increasing the number of steps of CoT can give more power than stochastic sampling. Finally, Xu & Sato (2026) also study stochastic CoT and implement randomized algorithms for approximate counting and sampling on that architecture.

2 NOTATION AND PRELIMINARIES

In this section we describe our notation, define our model of stochastic transformers and provide some basic definitions related to circuit complexity. See Appendices A, B and C for the details.

2.1 TRANSFORMERS

Transformers as language recognizers. A transformer is a machine that works over a finite alphabet Σ. It has a single tape from which it reads the input and onto which it appends additional tokens in an auto-regressive manner. In each step, the transformer maps the content of the tape to

[1] Namely, the amount of CoT steps grows as $O(\log n)$ where n is the size of the input.

a probability distribution over Σ, and then, depending on this distribution, it appends a new token onto the tape. The transformer iterates this process until it halts; these iterations are called **chain of thought** (Wei et al., 2022). There are two distinguished tokens called **decision symbols**. Once the transformer writes one of those, it halts. We say that a transformer accepts a word if, when it halts, the decision symbol printed is the accepting one (q_{accept}). Analogously, a transformer rejects a word if the decision symbol printed is the rejecting one (q_{reject}).

Note that we split the process of generating the next token into two separate steps. First, the computation of the probability distribution over the alphabet, which we call *distribution generation*. Second, the selection of a token based on the probability distribution, which we call *token selection*.

Distribution generation This process is described in Algorithm 1. It consists of four parts: a token embedding layer (TE), a position encoding layer (PE), an output linear layer (OUTPUT) and a stack of L identical decoder layers, where L is also called the depth of the model [2]. Each decoder layer has two sub-layers: a multi-head self-attention layer (ATTN) and a position-wise fully connected feed-forward network (FF). Each part and layer has its own trainable parameters. Then, we can define the distribution generation algorithm by its parameters $\theta = \{\theta_{\mathsf{TE}}, \theta_{\mathsf{PE}}, \{\theta^l_{\mathsf{ATTN}}, \theta^l_{\mathsf{FF}}\}_{l=0}^{L-1}, \theta_{\mathsf{OUTPUT}}\}$. The algorithm works over vectors of numbers of a certain dimension d.

Algorithm 1 Distribution generator

Input: Transformer parameter $\theta = (\theta_{\mathsf{TE}}, \theta_{\mathsf{PE}}, \{\theta^l_{\mathsf{ATTN}}, \theta^l_{\mathsf{FF}}\}_{l=0}^{L-1}, \theta_{\mathsf{OUTPUT}})$ and tokens of the tape $\alpha \in \Sigma^*$.
Output: Distribution $p_\theta(\cdot|\alpha)$.

1: $h_i^{(0)} \leftarrow \theta_{\mathsf{TE}}(\alpha_i) + \theta_{\mathsf{PE}}(i),\ \forall 1 \le i \le |\alpha|$
2: **for** $l = 0, \dots, L-1$ **do**
3: $\quad (h_1^{(l+0.5)}, \dots, h_{|\alpha|}^{(l+0.5)}) \leftarrow (h_1^{(l)}, \dots, h_{|\alpha|}^{(l)}) + \mathsf{ATTN}_{\theta^{(l)}_{\mathsf{ATTN}}}(h_1^{(l)}, \dots, h_{|\alpha|}^{(l)})$
4: $\quad h_i^{(l+1)} \leftarrow h_i^{(l+0.5)} + \mathsf{FF}_{\theta^{(l)}_{\mathsf{FF}}}(h_i^{(l+0.5)}), \forall 1 \le i \le |\alpha|$
5: **end for**
6: $p_\theta(\cdot|\alpha) \leftarrow \mathsf{OUTPUT}_{\theta_{\mathsf{OUTPUT}}}(h_{|\alpha|}^{(L)})$

Token selection To select the next token we consider two strategies: a deterministic one which returns the token from Σ with the largest probability, and a probabilistic one which samples a token according to the distribution.

Finite representation $\mathbb{F}_{e,s}$ In real models, the transformers typically work with 16- or 32-bit floating-point numbers, and there is active work in reducing their size. For this reason, in this paper we follow the decision of Li et al. (2024) to work with constant-precision transformers. Refer to Appendix B for a detailed definition.

Step notation We denote by $\mathsf{TF}(\alpha)$ the token outputted after one step of the computation of the transformer TF when the tape content is $\alpha \in \Sigma^*$. The auto-regressive token generation is defined recursively. The token outputted after i steps is $\mathsf{TF}^i(\alpha) := \mathsf{TF}^{i-1}(\alpha\mathsf{TF}(\alpha))$ with base case $\mathsf{TF}^1(\alpha) := \mathsf{TF}(\alpha)$. In other words, the output of the ith step is $\sigma_i = \mathsf{TF}^i(\alpha) = \mathsf{TF}(\alpha\sigma_1 \dots \sigma_{i-1})$ where $\sigma_j = \mathsf{TF}^j(\alpha) \in \Sigma$ for all $1 \le j < i$. We denote by $\mathsf{TF}^*(\alpha)$ the token written by the transformer when it halts, i.e., at the end of the computation, when the input is α. As said before, the transformer only halts when it prints a decision symbol. Therefore, $\mathsf{TF}^*(\alpha) \in \{q_{accept}, q_{reject}\}$.

Definition 2.1 (Deterministic transformer)**.** We call a transformer deterministic if it uses the deterministic token selector. We say that a family of deterministic transformers $\{\mathsf{TF}_n\}_{n\in\mathbb{N}}$ recognizes a language $\mathcal{L}$ when:

$$\alpha \in \mathcal{L} \iff \mathsf{TF}^*_{|\alpha|}(\alpha) = q_{accept}$$

When TF is a deterministic transformer, $\mathsf{TF}^i(\alpha)$ is an exact symbol for all $\alpha \in \Sigma^*$ and $i \in \mathbb{N}$. On the contrary, when the token selection process of the transformer samples the distribution, it becomes a random variable. Therefore, we introduce the following definition.

[2] See Appendix A for a more detailed definition of each component.

Definition 2.2 (Probabilistic transformer)**.** We call a transformer probabilistic if it uses the probabilistic token selector. We say that a family of probabilistic transformers $\{\mathsf{TF}_n\}_{n\in\mathbb{N}}$ recognizes a language $\mathcal{L}$ when:

$$\alpha \in \mathcal{L} \implies \Pr[\mathsf{TF}^*_{|\alpha|}(\alpha) = q_{accept}] > 2/3$$
$$\alpha \notin \mathcal{L} \implies \Pr[\mathsf{TF}^*_{|\alpha|}(\alpha) = q_{reject}] > 2/3$$

In general, given a family of transformers $\{\mathsf{TF}_n\}_{n\in\mathbb{N}}$ we use $d(n)$ to refer to the **embedding size** of transformer TF_n.

Determinizing step notation When working with probabilistic transformers, sampling from the distribution can be implemented using a binary string as a seed. Once the seed is fixed, the value of each step of CoT stops being a random variable. We extend the step notation to $\mathsf{TF}^i(\alpha, r_1 \dots r_i)$ for the token outputted by the i-th step when the seed is $r_1 \dots r_i$. We define it recursively as $\mathsf{TF}^i(\alpha, r_1 \dots r_i) := \mathsf{TF}^{i-1}(\alpha\mathsf{TF}(\alpha, r_1), r_2 \dots r_i)$. We use $\mathsf{TF}^1(\alpha, \lambda) := \mathsf{TF}(\alpha)$ as the base case, where λ is the empty word. Notice that $\Pr[\mathsf{TF}^i(\alpha) = \sigma] = \Pr_{r\in\{0,1\}^*}[\mathsf{TF}^i(\alpha, r) = \sigma]$ where r is chosen uniformly.

2.2 Boolean circuits

Boolean circuits Boolean circuits are a model of computation generalizing Boolean formulas. It is a simplified model of the silicon chips used to make modern computers. It is a natural model for nonuniform computation, as is the case for transformers; thus, most of our results on the expressiveness of transformers are stated in the language of circuit theory. A Boolean circuit is a procedural diagram showing how to derive an output from a binary input string by applying a sequence of basic Boolean operations OR ($\vee$), AND ($\wedge$), and NOT ($\neg$) on the input bits.

Randomized Boolean circuits These are Boolean circuits which are allowed to have random gates. These gates have fan-in 0, and their value is a uniformly distributed bit. This value is fixed once the evaluation of the circuit starts. Then, the evaluation of a randomized circuit C with input x can be thought of as evaluating a Boolean circuit C' with input xr, where r contains the values of the random gates. We say that a family of random circuits recognizes a language $\mathcal{L}$ when $\Pr[C_{|x|}(x) = \mathcal{L}(x)] \geq 2/3$ for every input x.

Randomized Boolean circuit classes We extend the definitions of standard Boolean circuit complexity classes (such as AC^0 or $\mathsf{P/Poly}$) to randomized circuits. We add the prefix BP to a Boolean circuit class to denote its randomized counterpart. Also, we add the suffix $[T(n)]$ to the class to denote the subset of languages solvable by randomized circuits using $O(T(n))$ random gates. For example, $\mathsf{BPAC}^0[\log n]$ is the class of problems recognizable by a family of polynomial-size randomized circuits with at most $O(\log n)$ random gates, constant depth and unbounded fan-in.

3 Complexity classes of bounded CoT

For any two functions $T(n)$ and $d(n)$ we define the complexity class $\mathsf{CoT}[T(n), d(n)]$ containing the set of languages recognizable by a deterministic family of transformers with constant depth, embedding size $d(n)$, and at most $T(n)$ steps of CoT. In the same way, we define the classes of languages recognized by probabilistic transformers.

Definition 3.1 ($\mathsf{CoT}[T(n), d(n)]$, $\mathsf{BPCoT}[T(n), d(n)]$)**.** A language $\mathcal{L}$ is in $\mathsf{CoT}[T(n), d(n)]$ (respectively, $\mathsf{BPCoT}[T(n), d(n)]$) if and only if there is a family of deterministic (respectively, probabilistic) transformers $\{\mathsf{TF}_n\}_{n\in\mathbb{N}}$ that recognizes $\mathcal{L}$ such that, if $d'(n)$ denotes the embedding size of each TF_n and $T'(n)$ denotes its number of CoT steps, then $d'(n) = O(d(n))$ and $T'(n) = O(T(n))$.

We note that these families are well-behaved. For example, they are closed under Boolean operations. Moreover, the probabilistic class allows for error reduction (namely, the constant chosen for the recognition probability is irrelevant as long as it is above $1/2$).

Proposition 3.1. $\mathsf{CoT}[T(n), d(n)]$ and $\mathsf{BPCoT}[T(n), d(n)]$ classes are closed under complement.

Proof Sketch. To construct a transformer that accepts the complement of another, it is enough to swap the probabilities of printing q_{accept} and q_{reject}, i.e., swap the corresponding rows of the OUTPUT matrix. Refer to Appendix F.1 for the complete proof. □

Proposition 3.2. $\mathsf{CoT}[T(n), d(n)]$ and $\mathsf{BPCoT}[T(n), d(n)]$ classes are closed under disjunction.

Proof Sketch. Given two transformers TF_1 and TF_2, we construct a third one that first runs TF_1 and then TF_2. Finally, it computes the disjunction of both outputs in one step of CoT. Our construction makes one step of CoT for each step made by TF_1 and TF_2 plus an extra one for the disjunction. Its embedding size is the sum of the embedding sizes of TF_1 and TF_2 multiplied by a constant factor. Refer to Appendix F.2 for the complete proof. □

Proposition 3.3. $\mathsf{BPCoT}[T(n), d(n)]$ is closed under error reduction, i.e., the constant used to bound the error does not affect the class.

Proof Sketch. We define a transformer that sequentially runs a probabilistic transformer TF r times and accepts the input word if and only if more than half of the runs accepted it. If TF has embedding size $d(n)$ and makes $T(n)$ steps of CoT, the constructed transformer has embedding size $O(rd(n))$ and makes $rT(n) + 1$ steps of CoT. Taking $r = O(1)$ is enough for correctness. Refer to Appendix F.3 for a complete proof. □

4 Extension of CoT bounds to BPCoT

4.1 Upper bound for transformer expressiveness

To begin this section we adapt Theorem 3.1 from Li et al. (2024) to probabilistic transformers. We prove that one step of a probabilistic, constant-depth, polynomial-embedding-dimension transformer can be simulated with a circuit in AC^0.

Theorem 4.1. $\mathsf{BPCoT}[1, \mathsf{poly}(n)] \subseteq \mathsf{AC}^0$

Li et al. (2024) gave a simulation of the distribution generator by an AC^0 circuit. Thus, we give a $\mathsf{BPAC}^0[1]$ circuit that simulates the sampling of a distribution computed by the distribution generator algorithm. Then, since $\mathsf{BPAC}^0[1] \subseteq \mathsf{AC}^0$ (Theorem C.4) we obtain the result.

Lemma 4.1. There is a $\mathsf{BPAC}^0[1]$ circuit that simulates the sampling of a probability distribution for the tokens defined in $\mathbb{F}_{e,s}$.

Proof Sketch. We define $|\Sigma|$ ranges in $[0, 1)$, one per token, each of length equal to the token's probability. These ranges can be computed by an AC^0 circuit because the probabilities are numbers in $\mathbb{F}_{e,s}$. Then, we use a seed of constant size to index the ranges. The seed is generated by the random gates and the ranges are indexable with an AC^0 circuit. Thus, the circuit outputs the token σ with probability $p(\sigma)$. Refer to Appendix G for a complete proof. □

Note that Theorem 4.1 follows immediately from Lemma 4.1 and the preceding discussion.

Now we show that, as in the case of deterministic transformers, $O(\log n)$ steps of CoT are not enough to go further than AC^0. Note that we can enumerate all $(|\Sigma|^{O(\log(n))} = n^{O(1)})$ possible chain-of-thought token sequences and $2^{O(\log(n))} = n^{O(1)}$ possible seeds. Then, we can check in parallel which of them are consistent (each check can be implemented in AC^0) and output their majority vote. We can compute this majority vote in AC^0 using the fact that the original transformer answered correctly for at least $2/3$ of the seeds. Therefore, at least $2/3$ of the consistent runs will agree, and under this promise majority is in AC^0 (see Theorem C.2).

Theorem 4.2. $\mathsf{BPCoT}[\log n, \mathsf{poly}(n)] \subseteq \mathsf{AC}^0$

Notice that Theorem 4.2 implies that transformers cannot solve problems like PARITY or MAJORITY without using a super-logarithmic number of steps of CoT.

Since one forward pass of constant-depth, polynomial-embedding-dimension, and constant-precision probabilistic transformers can be simulated by a circuit in $\mathsf{BPAC}^0[1]$, stacking the circuits of $\mathsf{BPAC}^0[1]$ for each CoT step gives a simulation for $\mathsf{poly}(n)$ steps. This circuit has polynomial size and uses a polynomial number of random bits; thus, we obtain Corollary 4.1 which will be used in later proofs.

Corollary 4.1. $\mathsf{BPCoT}[\mathsf{poly}(n), \mathsf{poly}(n)] \subseteq \mathsf{BPP/Poly}$

We now extend a result from Xu & Sato (2026) to our setting. This observation is similar to the one from Theorem 4.2.

Theorem 4.3. *For every integer $k \geq 1$, $\mathsf{BPCoT}[\log^k n, \mathsf{poly}(n)] \subseteq \mathsf{BPAC}^{k-1}$. More in general, any family of transformers in $\mathsf{BPCoT}[T(n), \mathsf{poly}(n)]$ with $T(n) = \mathsf{poly}(n)$ can be simulated by a family of randomized AC circuits of depth $O(1 + T(n)/\log n)$.*

Proof Sketch. To simulate a transformer that performs $O(\log^k n)$ steps of CoT in BPAC^{k-1} we split the $\log^k n$ steps in pieces of length $\log n$. Using arguments similar to the ones from the proof of Theorem 4.2 we can argue that, after fixing the next $\log n$ random seeds, the computation of the next $\log n$ steps of CoT is in AC^0. Composing this circuit $\log^{k-1} n$ times we obtain our AC^{k-1} circuit. Refer to Appendix I for the details. □

This proof holds for the uniform case. In the nonuniform one, we can remove the randomness.

Corollary 4.2. For every integer $k \geq 1$, $\mathsf{BPCoT}[\log^k n, \mathsf{poly}(n)] \subseteq \mathsf{TC}^{k-1}$.

Proof. By definition $\mathsf{BPAC}^j \subseteq \mathsf{BPTC}^j$. By Adleman's Theorem it holds that $\mathsf{BPTC}^j \subseteq \mathsf{TC}^j$. □

These results show that, under the common assumption that $\mathsf{BPP} \not\subseteq \mathsf{AC}$ to solve all problems in BPP, transformers need to perform an amount of CoT steps beyond $\log^k n$ for every k.

4.2 Lower bound for BPCoT classes

In this section we prove that constant-depth, constant-precision probabilistic transformers using T steps of CoT with $O(\log(n))$ embedding size can simulate a randomized circuit of size T. To do this, we extend the simulation of circuits of polynomial size by transformers given by Li et al. (2024) to the probabilistic setting. Their construction provides a transformer for each Boolean circuit such that, in the i-th step of the computation, it computes the i-th gate of the circuit (numbered in some topological order) using the input word of the transformer as input of the circuit. Then, the last token of the CoT is the value of the output gate of the circuit.

Theorem 4.4. *For any polynomial function $T : \mathbb{N}^+ \to \mathbb{N}^+$, $\mathsf{BPSIZE}(T(n)) \subseteq \mathsf{BPCoT}[T(n), \log(n)]$. Particularly, $\mathsf{BPP/Poly} \subseteq \mathsf{BPCoT}[\mathsf{poly}(n), \mathsf{poly}(n)]$.*

Proof Sketch. Let $\{C_n\}_{n\in\mathbb{N}}$ be a family of randomized circuits of polynomial size. We construct a family of probabilistic transformers $\{\mathsf{TF}_n\}_{n\in\mathbb{N}}$ such that each TF_n simulates C_n. As Definition C.12 states, a randomized circuit C_n accepts α when $\Pr_r[C'_n(\alpha, r) = 1] > 2/3$, that is, the deterministic circuit C'_n accepts (α, r) for more than $2/3$ of the possible values of r. Our construction first generates r at random with one step of CoT per bit of r, and then simulates the deterministic circuit C'_n with input αr. For simulating the deterministic circuit we use the construction presented in Theorem 3.3 from Li et al. (2024). Refer to Appendix H for the complete construction. □

4.3 Lower bounds based on sensitivity

The following result extends the lower bound based on average sensitivity from Theorem 4.2 and Corollary 4.3 of Chen et al. (2026) to probabilistic transformers. Intuitively, we apply their arguments after fixing the random bits, and then relate the sensitivity of these computations to that of the randomized result of our transformers.

For a Boolean function $f : \{0,1\}^n \to \{0,1\}$, its average sensitivity is defined as

$$\mathrm{AS}(f) = \sum_{i=1}^{n} \Pr[f(X) \neq f(X^{\oplus i})],$$

where X is sampled uniformly over $\{0,1\}^n$ and $x^{\oplus i}$ denotes x with its ith bit flipped.

The next lemma relates the number of CoT steps that a probabilistic transformer performs with the sensitivity of the function it computes.

Lemma 4.2. Let $\{\mathsf{TF}_n\}_{n\in\mathbb{N}}$ be a family of probabilistic transformers performing at most $T(n) = \mathsf{poly}(n)$ CoT steps. Then, there are constants $c, C > 0$ such that, for every $n \geq 2$ and every assignment r of the sampling bits,

$$\mathrm{AS}(\mathsf{TF}_n(\cdot, r)) \leq c(T(n)+1)(\log n)^C.$$

This bound on the sensitivity of the function $\mathsf{TF}_n(\cdot, r)$ arises from the limited way in which the information from x can be processed alongside $T(n)$ steps of CoT.

Proof Sketch. After fixing r, a circuit that decides, given an input x and a string t, whether a transformer TF outputs t as CoT given x as input and r as random seed can be implemented in AC^0. Thus, we can apply the techniques from Chen et al. (2026) to this function, and after averaging the result over all r we recover a bound for the probabilistic setting. Refer to Appendix J for the full proof. □

Now, to relate the sensitivity of a family of functions $\{f_n\}_{n\in\mathbb{N}}$ to the family of transformers computing it, we use the previous lemma, averaging over all possible values of r. We defer the details to the Appendix.

Theorem 4.5. *Let $f_n : \{0,1\}^n \to \{0,1\}$ be a family of non-constant functions computed by a probabilistic transformer family satisfying the assumptions of Lemma 4.2. Suppose further that, for a fixed $0 \leq \varepsilon < 1/2$ and every input x,*

$$\Pr_r[\mathsf{TF}_n(x, r) = f_n(x)] \geq 1 - \varepsilon.$$

Then, there are constants $c, C > 0$, depending on the transformer family, such that, for every $n \geq 2$,

$$(1 - 2\varepsilon)\,\mathrm{AS}(f_n) \leq c(T(n)+1)(\log n)^C.$$

In particular, if the language defined by $\{f_n\}$ belongs to $\mathsf{BPCoT}[T(n), \mathsf{poly}(n)]$, then

$$T(n) = \Omega\left(\frac{\mathrm{AS}(f_n)}{(\log n)^C}\right).$$

This bound is equal to the deterministic one whenever $\varepsilon = 0$, and for any error below $1/2$ the asymptotic growth order is the same. Note that the exponent C depends on the fixed circuit depth and is not a universal constant over all transformer families. We can use this result to obtain lower bounds for particular families of functions with high sensitivity, as it is done in Chen et al. (2026).

Corollary 4.3. Any family of probabilistic transformers as in Theorem 4.5 that computes PARITY with probability at least $2/3$ on every input requires $T(n) = \Omega\big(n/(\log n)^C\big)$ for some constant $C > 0$.

A family of functions can be hard to compute and still have low sensitivity: for instance, this can happen when only a subset of the instances is sensitive, in which case the average sensitivity is small. In such a situation, a standard way to obtain lower bounds is to construct a reduction from a more sensitive family, and then argue how the lower bounds extend. Such a technique is used in (Chattopadhyay & Santhanam, 2012, Lemma 3.11), which transfers compression lower bounds from PARITY to MAJORITY through adaptive queries. Here we consider AC^0 reductions, and develop machinery that, in particular, allows us to extend the lower bound from PARITY to REACHABILITY.

Proposition 4.1. Let $f_N : \{0,1\}^N \to \{0,1\}$ and $g_m : \{0,1\}^m \to \{0,1\}$ be families of Boolean functions. Suppose that $N : \mathbb{N} \to \mathbb{N}$ satisfies $m \le N(m) \le m^k$ for some constant k, and that there is a polynomial-size, constant-depth circuit family computing $\pi_m : \{0,1\}^m \to \{0,1\}^{N(m)}$ such that $g_m(z) = f_{N(m)}(\pi_m(z))$. Then, if a transformer family satisfying the assumptions of Theorem 4.5 computes f_N with error $\varepsilon < 1/2$ on every input and using at most $T(N) \le \mathsf{poly}(N)$ generated tokens, there are constants $c, C > 0$ such that

$$(1 - 2\varepsilon)\,\mathrm{AS}(g_m) \le c\big(T(N(m)) + 1\big)(\log m)^C.$$

Proof Sketch. The proof is derived by observing that, if $\mathsf{TF}_{N(m)}$ is the transformer computing $f_{N(m)}$, then after fixing the randomness r, the whole computation $\mathsf{TF}_{N(m)}(\pi_m(\cdot), r)$ can be analyzed using the same techniques from the proof of Lemma 4.2. □

We now apply this strategy to graph reachability using the classical reduction from parity. (which was already used to prove hard-attention CoT lower bounds (Amiri Bavandpour et al., 2025)). However, in our case we represent the graph with its adjacency matrix. See Appendix K for the details.

Corollary 4.4. Consider directed s-t reachability on n-vertex graphs encoded by their $N = n(n-1)$ adjacency bits, with s, t fixed and distinct. Any transformer family as in Theorem 4.5 that solves this problem with probability at least $2/3$ on every input requires, for some constant $C > 0$,

$$T(N) = \Omega\left(\frac{n}{(\log n)^C}\right) = \Omega\left(\frac{\sqrt{N}}{(\log N)^C}\right).$$

5 Relations between CoT and BPCoT

When defining probabilistic transformers, the relation between their expressive power and the expressive power of deterministic transformers becomes of interest. First, we show how to simulate a deterministic transformer with a probabilistic one.

Theorem 5.1. *A constant-depth, constant-precision deterministic transformer with embedding dimension $O(d(n))$ and using $T(n)$ steps of CoT can be simulated by a constant-depth, constant-precision probabilistic transformer with $O(d(n))$ embedding dimension and $O(T(n))$ CoT steps.*

Proof Sketch. For every deterministic transformer we can construct an equivalent one such that all the generated probabilities are saturated in the token that had the largest one in the original. Then, sampling the saturated distribution is the same as taking argmax over the original. This construction requires a constant blow-up in the embedding size. Refer to Appendix L for the details. □

Then, we immediately obtain the following:

Corollary 5.1. $\mathsf{CoT}[T(n), d(n)] \subseteq \mathsf{BPCoT}[T(n), d(n)]$ for all $T(\cdot)$ and $d(\cdot)$.

Li et al. (2024) proved that $\mathsf{CoT}[\mathsf{poly}(n), \mathsf{poly}(n)] = \mathsf{P/Poly}$. The same theorem can be deduced in the probabilistic setting from Theorem 4.4 and Corollary 4.1.

Corollary 5.2. $\mathsf{BPCoT}[\mathsf{poly}(n), \mathsf{poly}(n)] = \mathsf{BPP/Poly}$.

From Adleman's theorem it can be deduced that $\mathsf{P/Poly} = \mathsf{BPP/Poly}$. Then, we have:

Corollary 5.3. $\mathsf{CoT}[\mathsf{poly}(n), \mathsf{poly}(n)] = \mathsf{BPCoT}[\mathsf{poly}(n), \mathsf{poly}(n)]$.

This result states that, in the context of non-uniformity, the power of deterministic and probabilistic transformers is the same when using a polynomial number of CoT steps. Thus, unless we are studying fine-grained complexity, both models should yield the same results.

5.1 Restricting to uniform families

Notice that there are no restrictions on the constructability of the families of transformers of $\mathsf{CoT}[T(n), d(n)]$ and $\mathsf{BPCoT}[T(n), d(n)]$. In this section we analyze the uniform version of these classes, i.e., we require the families of transformers to be efficiently constructible.

We denote by $(\mathsf{Uniform})\mathsf{CoT}[T(n), d(n)]$ the class $\mathsf{CoT}[T(n), d(n)]$ restricted to problems that have a family of transformers such that there exists a deterministic Turing machine that runs in logarithmic space and, given the input length 1^n, outputs the transformer TF_n. In the same way we restrict $\mathsf{BPCoT}[T(n), d(n)]$ to $(\mathsf{Uniform})\mathsf{BPCoT}[T(n), d(n)]$.

Since the theorems proving upper and lower bounds for deterministic and probabilistic transformers give algorithms for the construction of the simulations, from Li et al. (2024)'s work and from Corollary 5.2 we can deduce the following corollary:

Corollary 5.4. $\mathsf{P} = (\mathsf{Uniform})\mathsf{CoT}[\mathsf{poly}(n), \mathsf{poly}(n)]$, $\mathsf{BPP} = (\mathsf{Uniform})\mathsf{BPCoT}[\mathsf{poly}(n), \mathsf{poly}(n)]$

Since it is not known whether $\mathsf{P} = \mathsf{BPP}$, we cannot deduce the uniform version of Corollary 5.3; we can only prove:

Theorem 5.2. $(\mathsf{Uniform})\mathsf{CoT}[\mathsf{poly}(n), \mathsf{poly}(n)] = (\mathsf{Uniform})\mathsf{BPCoT}[\mathsf{poly}(n), \mathsf{poly}(n)]$ *if and only if* $\mathsf{P} = \mathsf{BPP}$.

Although $\mathsf{P} = \mathsf{BPP}$ is an open question in complexity theory, this theorem is an interesting result. It implies that, when restricted to uniform families of transformers, the constrained access to randomness that transformers have is equivalent in expressive power to the unrestricted access allowed to probabilistic polynomial-time Turing machines. This result suggests that studying probabilistic transformers is particularly relevant in fine-grained complexity, given the widely held (though unproven) belief that randomness yields a polynomial advantage.

We show that non-uniformity in deterministic transformers can simulate randomness in uniform probabilistic transformers. We can relate Corollary 5.4 and Li et al. (2024)'s result by Adleman's Theorem and conclude Theorem 5.3:

Theorem 5.3. $(\mathsf{Uniform})\mathsf{BPCoT}[\mathsf{poly}(n), \mathsf{poly}(n)] \subseteq \mathsf{CoT}[\mathsf{poly}(n), \mathsf{poly}(n)]$

6 Conclusions

In this paper, we formalized the complexity classes associated with bounded CoT in probabilistic transformers under constant-depth and constant-precision assumptions. We established basic robustness properties of these classes, including closure under Boolean operations and error reduction. Along the way, our constructions also established the corresponding closure properties for the deterministic CoT classes. We then extended several arguments developed for deterministic transformers to the probabilistic setting, showing that many upper and lower bounds remain valid under sampling.

In particular, we showed that the problems solvable by probabilistic transformers with logarithmically many CoT steps are included in AC^0, even with polynomial embedding dimension, and that $O(\log^k n)$ steps admit simulation in BPAC^{k-1}. Conversely, randomized circuits of polynomial size $T(n)$ can be simulated using $O(T(n))$ CoT steps and embedding dimension $O(\log n)$. We also extended lower bounds based on average sensitivity to our setting, obtaining nearly linear CoT requirements for PARITY and a sublinear for REACHABILITY. Finally, we characterized the computational power of polynomially bounded CoT: nonuniform probabilistic transformers capture P/poly, while polynomial-time-uniform probabilistic and deterministic families capture BPP and P, respectively. Thus, their equality in the uniform setting is equivalent to $\mathsf{P} = \mathsf{BPP}$.

Several questions remain open. Can we identify practically useful tasks for which probabilistic transformers require substantially fewer CoT steps than deterministic transformers with comparable embedding dimension and precision? Which additional lower bound techniques extend to the stochastic setting? Another important direction is to incorporate layer normalization into our probabilistic model, which would bring the abstraction closer to practical transformer architectures. Beyond the architecture itself, modern LLMs can solve problems such as parity by using tool calls to other systems (such as running a piece of code). One could introduce this resource into the framework by giving the transformers access to an oracle which is executed after printing a particular symbol in the CoT. Understanding the strength of this model and its dependence on the strength of the oracle could shed light on the current capacity of LLMs.

AI disclosure We used LLMs (in particular, ChatGPT 5.6 Sol and ChatGPT 6 Astra) to aid and polish writing, and for retrieval and discovery or related literature.

REFERENCES

Leonard M. Adleman. Two theorems on random polynomial time. In *19th Annual Symposium on Foundations of Computer Science*, pp. 75–83, 1978. doi: 10.1109/SFCS.1978.37.

Alireza Amiri Bavandpour, Xinting Huang, Mark Rofin, and Michael Hahn. Lower bounds for chain-of-thought reasoning in hard-attention transformers. In *Proceedings of the 42nd International Conference on Machine Learning*, volume 267 of *Proceedings of Machine Learning Research*, pp. 3274–3306, 2025. URL https://proceedings.mlr.press/v267/bavandpour25a.html.

Bowen Baker, Joost Huizinga, Leo Gao, Zehao Dou, Melody Y Guan, Aleksander Madry, Wojciech Zaremba, Jakub Pachocki, and David Farhi. Monitoring reasoning models for misbehavior and the risks of promoting obfuscation. *arXiv preprint arXiv:2503.11926*, 2025.

Shaked Brody, Uri Alon, and Eran Yahav. On the expressivity role of LayerNorm in transformers' attention. In *Findings of the Association for Computational Linguistics: ACL 2023*, pp. 14211–14221, 2023. doi: 10.18653/v1/2023.findings-acl.895. URL https://aclanthology.org/2023.findings-acl.895/.

Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901, 2020.

Arkadev Chattopadhyay and Rahul Santhanam. Lower bounds on interactive compressibility by constant-depth circuits. In *2012 IEEE 53rd Annual Symposium on Foundations of Computer Science*, pp. 619–628. IEEE, 2012.

Lijie Chen, Zhiyuan Li, Kaifeng Lyu, Binghui Peng, and Hongxun Wu. The information bottleneck of chain-of-thought and how latent cot overcomes it, 2026. URL https://openreview.net/forum?id=cCIdxLoLJ5.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde De Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.

David Chiang. Transformers in DLOGTIME-uniform TC^0. *Transactions on Machine Learning Research*, 2025. URL https://openreview.net/forum?id=ZA7D4nQuQF.

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 (Volume 1: Long Papers)*, pp. 7654–7664, 2022. doi: 10.18653/v1/2022.acl-long.527. URL https://aclanthology.org/2022.acl-long.527/.

Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, Maarten Bosma, Gaurav Mishra, Adam Roberts, Paul Barham, Hyung Won Chung, Charles Sutton, Sebastian Gehrmann, et al. Palm: Scaling language modeling with pathways. *Journal of machine learning research*, 24(240):1–113, 2023.

Zihang Dai, Zhilin Yang, Yiming Yang, Jaime G Carbonell, Quoc Le, and Ruslan Salakhutdinov. Transformer-xl: Attentive language models beyond a fixed-length context. In *Proceedings of the 57th annual meeting of the association for computational linguistics*, pp. 2978–2988, 2019.

Scott Emmons, Erik Jenner, David K Elson, Rif A Saurous, Senthooran Rajamanoharan, Heng Chen, Irhum Shafkat, and Rohin Shah. When chain of thought is necessary, language models struggle to evade monitors. *arXiv preprint arXiv:2507.05246*, 2025.

Guhao Feng, Bohang Zhang, Yuntian Gu, Haotian Ye, Di He, and Liwei Wang. Towards revealing the mystery behind chain of thought: A theoretical perspective. In *Advances in Neural Information Processing Systems*, volume 36, 2023. URL https://arxiv.org/abs/2305.15408.

Merrick Furst, James B Saxe, and Michael Sipser. Parity, circuits, and the polynomial-time hierarchy. *Mathematical systems theory*, 17(1):13–27, 1984.

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/.

IEEE. Ieee standard for floating-point arithmetic. *IEEE Std 754-2008*, pp. 1–70, 2008. doi: 10.1109/IEEESTD.2008.4610935.

John Jumper, Richard Evans, Alexander Pritzel, Tim Green, Michael Figurnov, Olaf Ronneberger, Kathryn Tunyasuvunakool, Russ Bates, Augustin Žídek, Anna Potapenko, et al. Highly accurate protein structure prediction with alphafold. *nature*, 596(7873):583–589, 2021.

Takeshi Kojima, Shixiang Shane Gu, Machel Reid, Yutaka Matsuo, and Yusuke Iwasawa. Large language models are zero-shot reasoners. In *Advances in Neural Information Processing Systems*, volume 35, 2022. URL https://arxiv.org/abs/2205.11916.

Tomek Korbak, Mikita Balesni, Elizabeth Barnes, Yoshua Bengio, Joe Benton, Joseph Bloom, Mark Chen, Alan Cooney, Allan Dafoe, Anca Dragan, et al. Chain of thought monitorability: A new and fragile opportunity for ai safety. *arXiv preprint arXiv:2507.11473*, 2025.

Zhiyuan Li, Hong Liu, Denny Zhou, and Tengyu Ma. Chain of thought empowers transformers to solve inherently serial problems. In *International Conference on Learning Representations*, volume 2024, pp. 11911–11943, 2024.

Zeming Lin, Halil Akin, Roshan Rao, Brian Hie, Zhongkai Zhu, Wenting Lu, Nikita Smetanin, Robert Verkuil, Ori Kabeli, Yaniv Shmueli, et al. Evolutionary-scale prediction of atomic-level protein structure with a language model. *Science*, 379(6637):1123–1130, 2023.

William Merrill and Ashish Sabharwal. The parallelism tradeoff: Limitations of log-precision transformers. *Transactions of the Association for Computational Linguistics*, 11:531–545, 2023. 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 *International Conference on Learning Representations*, 2024. URL https://arxiv.org/abs/2310.07923.

Parsa Mirtaheri, Ezra Edelman, Samy Jelassi, Eran Malach, and Enric Boix-Adserà. Let me think! a long chain-of-thought can be worth exponentially many short ones. In *Advances in Neural Information Processing Systems*, volume 38, 2025. URL https://arxiv.org/abs/2505.21825.

Franz Nowak, Anej Svete, Alexandra Butoi, and Ryan Cotterell. On the representational capacity of neural language models with chain-of-thought reasoning. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 12510–12548, 2024. doi: 10.18653/v1/2024.acl-long.676. URL https://aclanthology.org/2024.acl-long.676/.

Catherine Olsson, Nelson Elhage, Neel Nanda, Nicholas Joseph, Nova DasSarma, Tom Henighan, Ben Mann, Amanda Askell, Yuntao Bai, Anna Chen, et al. In-context learning and induction heads. *arXiv preprint arXiv:2209.11895*, 2022.

Avishay Tal. Tight bounds on the fourier spectrum of ac0. In *32nd Computational Complexity Conference (CCC 2017)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2017.

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*, volume 30, 2017. URL https://arxiv.org/abs/1706.03762.

Emanuele Viola. On approximate majority and probabilistic time. *Computational Complexity*, 18 (3):337, 2009.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. Chain-of-thought prompting elicits reasoning in large language models. *Advances in neural information processing systems*, 35:24824–24837, 2022.

Kevin Xu and Issei Sato. A formal comparison between chain of thought and latent thought. In *International Conference on Machine Learning*, 2026. URL https://arxiv.org/abs/2509.25239. Camera-ready version; first arXiv version appeared in 2025.

Andy Yang, Christopher Watson, Anton Xue, Satwik Bhattamishra, Jose Llarena, William Merrill, Emile Dos Santos Ferreira, Anej Svete, and David Chiang. The transformer cookbook, 2025. URL https://arxiv.org/abs/2510.00368.

A TRANSFORMER LAYERS

As stated before, the **distribution generation** is a mapping of the content of the tape to a probability distribution over Σ. There are different algorithms for this process in the literature. All transformers considered in this work employ the one presented in Li et al. (2024). This algorithm consists of four parts: a token embedding layer (TE), a position encoding layer (PE), an output linear layer (OUTPUT) and a stack of L identical decoder layers, where L is also called the depth of the model. Each decoder layer has two sub-layers: a multi-head self-attention layer (ATTN) and a position-wise fully-connected feed-forward network (FF). Each part and layer has its own trainable parameters. Then, we can define the distribution generation algorithm by its parameters $\theta = \{\theta_{\mathsf{TE}}, \theta_{\mathsf{PE}}, \{\theta^l_{\mathsf{ATTN}}, \theta^l_{\mathsf{FF}}\}_{l=0}^{L-1}, \theta_{\mathsf{OUTPUT}}\}$. The algorithm works over vectors of a certain dimension d. This dimension depends on the length of the input word.

Most prior work only considers transformers that operate over an infinite tape. This is far from real models; thus, the transformers studied in this work operate over a finite tape. For each input length n, the size of the tape is defined as $\mathrm{budget}(n) + n$, for some function $\mathrm{budget}\colon \mathbb{N} \to \mathbb{N}$. It is important to remark that the size of the tape bounds the number of CoT steps that the transformer makes: for an input word of length n, no computation can take more than $\mathrm{budget}(n)$ steps, because every step writes a token on the tape. We use $d(n)$ to refer to the **embedding size** when the input word has length n.

We now describe each part of the transformer's architecture in detail:

Token Embedding: It is a mapping from Σ to vectors in $\mathbb{R}^d$. This mapping is defined by $\theta_{\mathsf{TE}} \in \mathbb{R}^{d\times|\Sigma|}$. For all $\sigma \in \Sigma$, we denote by $\theta_{\mathsf{TE}}(\sigma) \in \mathbb{R}^d$ the mapping of token σ. We encode $\theta_{\mathsf{TE}}(\sigma)$ as the column of θ_{TE} indexed by σ.

Position Encoding: It is a mapping from the positions of the tape to vectors in $\mathbb{R}^d$. This mapping is defined by $\theta_{\mathsf{PE}} \in \mathbb{R}^{d\times(\mathrm{budget}(n)+n)}$. For all positions of the tape $1 \le i \le \mathrm{budget}(n) + n$, we denote by $\theta_{\mathsf{PE}}(i) \in \mathbb{R}^d$ the mapping of position i. As in the token embedding, we encode $\theta_{\mathsf{PE}}(i)$ as the i-th column of θ_{PE}.

Self-Attention Layer: For every $a \in \mathbb{N}$, it is a mapping from $(\mathbb{R}^d)^a$ to $(\mathbb{R}^d)^a$. Given parameters $\theta_{\mathsf{ATTN}} = (W_Q, W_K, W_V, W_O) \in \mathbb{R}^{d\times d} \times \mathbb{R}^{d\times d} \times \mathbb{R}^{d\times d} \times \mathbb{R}^{d\times d}$, the layer is defined by Algorithm 2. We define this layer as a single-head attention layer, since a multi-head attention layer with any constant number of heads can be simulated with multiple single-head attention layers (Yang et al., 2025).[3]

Algorithm 2 Causal Self-Attention, ATTN

Input: Parameters $\theta_{\mathsf{ATTN}} = (W_Q, W_K, W_V, W_O)$ and embedding $h = (h_1, \ldots, h_a) \in \mathbb{R}^d \times \cdots \times \mathbb{R}^d$.

Output: Embedding $h' = (h'_1, \ldots, h'_a) := \mathsf{ATTN}_{\theta_{\mathsf{ATTN}}}(h_1, \ldots, h_a)$.

1: $q_i := W_Q h_i,\ k_i := W_K h_i,\ v_i := W_V h_i,\ \forall i \in [1, \ldots, a]$
2: $s_i := \mathrm{softmax}(\langle q_i, k_1\rangle, \ldots, \langle q_i, k_i\rangle) \| (0, \ldots, 0) \in \mathbb{R}^a,\ \forall i \in [1, \ldots, a]$ ▷ padded with $a - i$ zeros
3: $h'_i := W_O \sum_{j=1}^{a} (s_i)_j v_j,\ \forall i \in [1, \ldots, a]$

Feed-Forward Network: It is a mapping from $\mathbb{R}^d$ to $\mathbb{R}^d$. Given $\theta_{\mathsf{FF}} = (W_1, b_1, W_2, b_2)$ in $\mathbb{R}^{d\times d} \times \mathbb{R}^d \times \mathbb{R}^{d\times d} \times \mathbb{R}^d$, the layer is defined as $\mathsf{FF}_{\theta_{\mathsf{FF}}}(h) := W_2\ \mathrm{relu}(W_1 h + b_1) + b_2$, where $\mathrm{relu} : \mathbb{R}^d \to \mathbb{R}^d$ replaces each coordinate x_i by $\max(0, x_i)$.

Output Layer: It is a mapping from $\mathbb{R}^d$ to a probability distribution over Σ. Given $\theta_{\mathsf{OUTPUT}} \in \mathbb{R}^{|\Sigma|\times d}$, we define it as $\mathsf{OUTPUT}_{\theta_{\mathsf{OUTPUT}}}(h) := \mathrm{softmax}(\theta_{\mathsf{OUTPUT}} h)$.

All these parts come together in the distribution generation process described in Algorithm 1.

[3] The matrices are named Q for query, K for key, V for value and O for output. It is called query because q_i represents a question of how much the i-th vector should attend to the others in a specific setting (determined by W_Q). The k_j vector answers said question such that $\langle q_i, k_j\rangle$ is large if the j-th vector is relevant to the i-th. Thus, the key matrix is used to compute the relevance. The value matrix has the role of extracting the information that the j-th vector should give to the i-th. Finally, the output matrix is just a linear post-processing.

The OUTPUT layer and the ATTN layer use the function $\mathrm{softmax}\colon \mathbb{R}^n \to \mathbb{R}^n$ defined as $\mathrm{softmax}(x)_i = \exp(x_i)/\sum_{j=1}^{n} \exp(x_j)$ (see Appendix B for its finite-precision version).

B FINITE REPRESENTATION OF NUMBERS

Intuitively, we represent the numbers as a tuple $\langle$sign, significand, exponent$\rangle$ with a fixed number of bits for each coordinate. The tuple is interpreted as sign $\cdot$ significand $\cdot\, 2^{\text{exponent}}$. For example, the number 8 is represented as $\langle +, 1, 3\rangle$. When the number of bits for the significand and the exponent is fixed, some numbers are not representable. For example, with only 2 bits of significand and 3 of exponent, the number 9 cannot be represented exactly. We can represent $8 = 1 \cdot 2^3$ and $12 = 3 \cdot 2^2$; since 9 is closer to 8, when an arithmetic operation should output 9 we output 8.

Below we give a formal definition of floating-point numbers and of the rounding operation. We use $\phi(a) = \sum_{i=1}^{k} 2^{k-i} a_i$ to denote the integer value of the binary string $a \in \{0,1\}^k$ for any $k \in \mathbb{N}^+$.

Definition B.1 (Floating-point Representation)**.** Let e be the number of bits for exponents and $2s$ be the number of bits for significand. An $(e + 2s + 1)$-bit binary string $a = (a_1, a_2, \ldots, a_{e+2s+1}) \in \{0,1\}^{e+2s+1}$ is a floating-point binary representation of number $\phi_{e,s}(a) \triangleq \text{sign}(a) \cdot 2^{\text{exponent}(a)} \cdot \text{significand}(a)$ with e-bit exponent and $2s$-precision, where the sign is $\text{sign}(a) \triangleq 2a_1 - 1$, the significand is $\text{significand}(a) \triangleq 2^{-s}\phi(a_{2:2s+1})$, and the exponent is $\text{exponent}(a) \triangleq \phi(a_{2s+2:2s+e+1}) - 2^{\max(0,e-1)}$. We define the set of numbers expressible with e bits of exponent and $2s$ bits of significand as $\mathbb{F}_{e,s}$. We define $B^+_{e,s}$ as the maximum number in $\mathbb{F}_{e,s}$. Similarly, we define $B^-_{e,s}$ as the minimum. Since in this work we only analyze constant-precision transformers, we will omit the subscripts of B^+ and B^-.

Definition B.2 (Correct Rounding)**.** For any $x \in \mathbb{R}$, we define correct rounding $\text{round}(x, \mathbb{F}_{e,s})$ as the closest number to x in $\mathbb{F}_{e,s}$. We break ties by picking the one with the smaller absolute value. In particular, we denote the rounding operation with e-bit exponent and $2s$-bit precision by $\text{round}_{e,s}(\cdot) \triangleq \text{round}(\cdot, \mathbb{F}_{e,s})$, which is also denoted by $[\cdot]_{e,s}$ for convenience. We extend the definition of round and $\text{round}_{e,s}$ to vector inputs by applying rounding coordinate-wise.

This notion of floating-point number simplifies the IEEE 754 Standard for Floating-point Arithmetic (IEEE, 2008) by removing ∞ and $-\infty$. When the result of an operation is larger than B^+ (overflow), we always round it to B^+, and when it is smaller than B^-, we round it to B^-. For unary functions like $\exp(\cdot)$ and binary functions including addition, subtraction, multiplication, and division, we simply define their rounded version by rounding their outputs. Whenever division by 0 happens, we define the output as 0.[4]

Next, we define finite-precision summation over more than two numbers by decomposing it as a chain of rounded binary additions in a fixed order.

Definition B.3 (Summation with Iterative Rounding)**.** For any $e, s \in \mathbb{N}^+$, we define summation with iterative rounding to e-bit exponent and $2s$-bit precision as $\mathsf{sum}_{e,s} : \cup_{n\in\mathbb{N}^+} (\mathbb{F}_{e,s})^n \to \mathbb{F}_{e,s}$, where for any $n \in \mathbb{N}^+$ and $x \in (\mathbb{F}_{e,s})^n$,

$$\mathsf{sum}_{e,s}(x) \triangleq \left[\left[\left[\left[x_1 + x_2\right]_{e,s} + x_3\right]_{e,s} + \cdots + x_{n-1}\right]_{e,s} + x_n\right]_{e,s}.$$

We further define the following operations:

- Finite-precision inner product: $\langle x, y\rangle_{e,s} \triangleq \mathsf{sum}_{e,s}(x \odot y)$
- Finite-precision matrix product: $(A \times_{e,s} B)_{i,j} \triangleq \left\langle (A_{i,:})^\top, B_{:,j}\right\rangle_{e,s}$
- Finite-precision softmax: $\text{softmax}_{e,s}(x) \triangleq \left[[\exp(x)]_{e,s} \,/\mathsf{sum}_{e,s}([\exp(x)]_{e,s})\right]_{e,s}$

The saturation and the iterative rounding presented by Li et al. (2024) yield the following properties, which we use repeatedly (sums are evaluated from left to right):

[4] Notice that the only division is in the softmax. If it occurs in the attention mechanism, it is reasonable to give an attention score of 0 to every vector. If it occurs in the output layer, it means that all symbols have an extremely small probability; therefore, it is reasonable to assign the same probability to all of them.

- $\exp(B^-) = 0$
- $\exp(B^+) = B^+$
- $B^- \cdot B^+ = B^-$
- $B^+ \cdot B^+ = B^+$
- $\forall x \in \mathbb{F}_{e,s}$ it holds that $x + 0 = x$
- $\forall x \in \mathbb{F}_{e,s}$ it holds that $x - x = 0$
- $\forall x \in \mathbb{F}_{e,s}$ it holds that $x + B^+ + B^+ + B^- = 0$

C PRELIMINARIES ON COMPLEXITY THEORY

Definition C.1 (Language)**.** In this work we consider the following notion of problems: given a sequence of input tokens, output a token as the answer. Mathematically, given a vocabulary Σ, we call a mapping $\mathcal{L} : \cup_{k\in\mathbb{N}^+}\Sigma^k \to \Sigma$ a problem. If the answer is always q_{accept} or q_{reject}, we call $\mathcal{L}$ a decision problem. Such $\mathcal{L}$ is also called a **language**.

C.1 BOOLEAN CIRCUITS

A Boolean circuit is a procedural diagram showing how to derive an output from a binary input string by applying a sequence of basic Boolean operations OR ($\vee$), AND ($\wedge$), and NOT ($\neg$) on the input bits.

Though the standard definition of circuit complexity only deals with binary strings, given any finite vocabulary Σ, we can always replace each token in Σ by its binary representation, and the length of the input only blows up by a constant factor. Therefore, we can extend existing complexity classes to arbitrary finite vocabularies naturally.

Definition C.2 (Boolean circuits)**.** For every $n \in \mathbb{N}$, an n-input, single-output **Boolean circuit** C is a directed acyclic graph with n sources (vertices with no incoming edges) and one sink (vertex with no outgoing edges). All nonsource vertices are called gates and are labeled with one of $\wedge$, $\vee$ or $\neg$ (i.e., the logical operations AND, OR, and NOT). The vertices labeled with $\vee$ and $\wedge$ have fan-in (i.e., number of incoming edges) equal to 2 and the vertices labeled with $\neg$ have fan-in 1. The size of C, denoted by $|C|$, is the number of vertices in it.

If C is a Boolean circuit, and $x \in \{0,1\}^n$ is some input, then the output of C on x, denoted by $C(x)$, is defined in the natural way. More formally, for every vertex v of C, we give it a value $val(v)$ as follows: If v is the i-th input vertex then $val(v) = x_i$, and otherwise $val(v)$ is defined recursively by applying v's logical operation to the values of the vertices with an edge into v. The output $C(x)$ is the value of the output vertex.

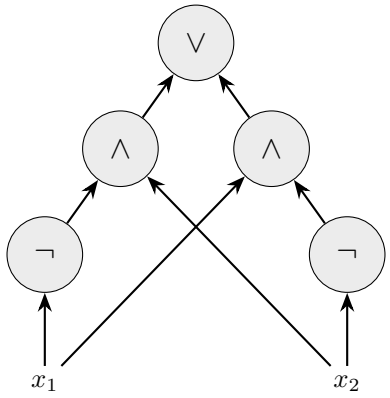

Figure 1: A circuit C computing the XOR function (i.e., $C(x_1, x_2) = 1$ iff $x_1 \neq x_2$).

Definition C.3 (Circuit families and language recognition)**.** Let $T : \mathbb{N} \to \mathbb{N}$ be a function. A $T(n)$-size **circuit family** is a sequence $\{C_n\}_{n\in\mathbb{N}}$ of Boolean circuits, where C_n has n inputs and a single output, and its size $|C_n| \leq T(n)$ for every n. We say that a language $\mathcal{L}$ is in $\mathsf{SIZE}(T(n))$ if there exists a $T(n)$-size circuit family $\{C_n\}_{n\in\mathbb{N}}$ such that for every $n \in \mathbb{N}$ and every $x \in \{0,1\}^n$, $x \in \mathcal{L} \iff C_n(x) = 1$.

Interesting complexity classes arise when we consider "small" circuits, such as the class of languages recognized by polynomial-size families.

Definition C.4 ($\mathsf{P/Poly}$)**.** $\mathsf{P/Poly}$ is the class of languages that are decidable by polynomial-sized circuit families. That is, $\mathsf{P/Poly} = \cup_{c\geq 0}\mathsf{SIZE}(n^c)$.

The class $\mathsf{P/Poly}$ fits rather awkwardly in the complexity world since it contains undecidable languages such as the unary halting problem[5]. The root of the problem is that for a language $\mathcal{L}$ to be in $\mathsf{P/Poly}$, it suffices that a circuit family for $\mathcal{L}$ exists even if we have no way of actually constructing the circuits. Thus, it may be fruitful to restrict attention to circuits that can actually be built, say using a fairly efficient Turing machine.

Definition C.5 (P-uniform circuit families)**.** A circuit family $\{C_n\}_{n\in\mathbb{N}}$ is P-uniform if there is a polynomial-time Turing machine that on input 1^n outputs the description of the circuit C_n.

Boolean circuits are a good model to represent parallel computation. The **depth** of a circuit is the length of the longest directed path from an input node to the output node.

Definition C.6 (NC)**.** For every i, a language $\mathcal{L}$ is in NC^i if $\mathcal{L}$ can be decided by a family of circuits $\{C_n\}_{n\in\mathbb{N}}$ where C_n has $\mathsf{poly}(n)$ size and depth $O(\log^i(n))$. The class NC is $\cup_{i\geq 1}\mathsf{NC}^i$.

When studying the depth of a circuit, the fan-in of the gates becomes of interest. A gate with polynomial fan-in can be simulated by gates of fan-in 2 with a logarithmic blow-up in depth. This motivates the study of circuits with unbounded fan-in.

Definition C.7 (AC)**.** The class AC^i is defined similarly to NC^i except gates are allowed to have unbounded fan-in (i.e., the OR and AND gates can be applied to more than two bits). The class AC is $\cup_{i\geq 0}\mathsf{AC}^i$.

AC^0 circuits can perform a number of interesting operations. For example, addition and multiplication of numbers of fixed precision can be done with AC^0 circuits. Also, the equality of two strings of bits can be decided in AC^0. From the ability to compute equality it follows that any function f that consumes only $O(\log(n))$ bits of the input can be computed as a subcircuit in AC^0. The circuit computing f enumerates the result for each of the possible inputs, and outputs the one that corresponds to the branch equal to the input. If k denotes the number of input bits of f, then f can be represented by the formula $\vee_{i=0}^{2^k-1}(f(i)\wedge[i=x])$. Observe that the formula represents a circuit of constant depth. Also note that the formula has size $O(2^k)$; then, if $k\in O(\log(n))$, the circuit has polynomial size; and, therefore, it is in AC^0.

One of the main results of circuit complexity is that the $\mathsf{MAJORITY}$ problem[6] cannot be computed by AC^0 circuits (Furst et al., 1984).

Theorem C.1. $\mathsf{MAJORITY}\notin\mathsf{AC}^0$

This theorem motivates the study of threshold circuits, i.e., circuits allowed to have $\mathsf{MAJORITY}$ gates.

Definition C.8 (TC)**.** The class TC^i is defined similarly to AC^i except the circuits can have gates OR, AND, NOT and $\mathsf{MAJORITY}$ of unbounded fan-in. The class TC is $\cup_{i\geq 0}\mathsf{TC}^i$.

Following Theorem C.1, a family of variations of $\mathsf{MAJORITY}$ began to be studied.

Definition C.9 ($\mathsf{MAJORITY}$ with promise)**.** For every $e\in[0,1/2)$ we define the problem **Majority with promise** (MAJ_e) as $\mathsf{MAJORITY}$ but restricted to inputs that have more than $(1/2+e)n$ 1s or more than $(1/2+e)n$ 0s, where n is the length of the input.

Observe that MAJ_0 is $\mathsf{MAJORITY}$ restricted to inputs with a different number of 1s and 0s.

Theorem C.2 (Viola (2009))**.** *For all $e>0$, there is a circuit in AC^0 recognizing MAJ_e.*

C.1.1 Randomized Boolean circuits

In this work we also consider randomized Boolean circuits. These are Boolean circuits that are allowed to have random gates. These gates have fan-in 0, and their value is a uniformly distributed

[5] $\mathsf{UHALT}=\{1^n : n$'s binary expansion encodes a pair $\langle M,x\rangle$ such that M halts on input $x\}$

[6] $\mathsf{MAJORITY}=\left\{x : \frac{\sum_{i=1}^{|x|}x_i}{|x|}>\frac{1}{2}\right\}$, i.e., the words with more 1s than 0s.

bit, independent of the other random gates. This value is fixed once the evaluation of the circuit starts. Then, the evaluation of a randomized circuit C with input x can be thought of as evaluating a Boolean circuit C' with input xr, where r are the values of the random gates.

Definition C.10 (Randomized Boolean circuits)**.** A **randomized Boolean circuit** C is a directed acyclic graph with $n+m$ sources and one sink. All nonsource vertices are labeled as $\wedge$, $\vee$ or $\neg$ (i.e., the logical operations AND, OR, and NOT). The vertices labeled with $\vee$ and $\wedge$ have fan-in (i.e., number of incoming edges) equal to 2 and the vertices labeled with $\neg$ have fan-in 1. The size of C, denoted by $|C|$, is the number of vertices in it. The source vertices are labeled either as input (n of them) or as random (m of them).

If C is a randomized Boolean circuit, and $x \in \{0,1\}^n$ is some input, then the output of C on x, denoted by $C(x)$, is defined as a random variable in the following way. For every vertex v of C, we give it a value $val(v)$ as follows: If v is the i-th input vertex, then $val(v) = x_i$; if v is labeled as random, then $val(v)$ is a uniform random bit, independent of the other random vertices; and otherwise $val(v)$ is defined recursively by applying v's logical operation to the values of the vertices with an edge into v. The output $C(x)$ is the value of the output vertex. See Figure 2 for a visual representation.

In general, we say that a family of randomized circuits recognizes a language $\mathcal{L}$ when $\Pr[C_{|x|}(x) = \mathcal{L}(x)]$ is high (identifying q_{accept} with 1 and q_{reject} with 0).

Definition C.11 (Randomized circuit families and language recognition)**.** Let $T : \mathbb{N} \to \mathbb{N}$ be a function. A $T(n)$-size **randomized circuit family** is a sequence $\{C_n\}_{n\in\mathbb{N}}$ of randomized Boolean circuits, where C_n has n inputs and a single output, and its size $|C_n| \leq T(n)$ for every n. We say that a language $\mathcal{L}$ is in $\mathsf{BPSIZE}(T(n))[f(n)]$ if there exists a $T(n)$-size randomized circuit family $\{C_n\}_{n\in\mathbb{N}}$ with $O(f(n))$ random gates such that for every $n \in \mathbb{N}$ and every $x \in \{0,1\}^n$, $\Pr[C_n(x) = \mathcal{L}(x)] \geq 2/3$.[7]

Since every randomized Boolean circuit can be interpreted as a Boolean circuit with the random bits padded to the input, we can also define recognition in the following way:

Definition C.12. We say that a language $\mathcal{L}$ is in $\mathsf{BPSIZE}(T(n))[f(n)]$ if there exists a $T(n)$-size family of deterministic circuits $\{C'_n\}_{n\in\mathbb{N}}$, where C'_n has $n + f(n)$ inputs, such that for every $n \in \mathbb{N}$ and every $x \in \{0,1\}^n$, $\Pr_{r\in\{0,1\}^{f(n)}}[C'_n(xr) = \mathcal{L}(x)] \geq 2/3$.

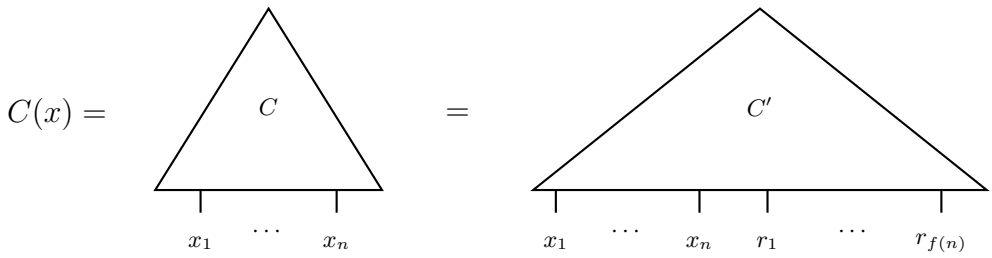

Figure 2: Circuit C padded with extra random inputs $r_1, \ldots, r_{f(n)}$ to form C'.

In general, we drop the bound on the number of random bits when it is not of interest.

Definition C.13. $\mathsf{BPSIZE}(T(n)) = \mathsf{BPSIZE}(T(n))[T(n)]$.

Definition C.14. Similarly to P/Poly, we define BPP/Poly as the class of languages that are decidable by polynomial-sized randomized circuit families. That is, $\mathsf{BPP/Poly} = \cup_{c\geq 0}\mathsf{BPSIZE}(n^c)$.

We extend the definitions of NC, NC^i, AC, AC^i, TC, TC^i (both uniform and non-uniform) to randomized Boolean circuits in the natural way. We denote by $\mathcal{C}[f(n)]$ the class $\mathcal{C}$ restricted to $O(f(n))$ random gates.

Observe that a Boolean circuit is a randomized Boolean circuit without random gates.

[7] As in many probabilistic complexity classes, the constant 2/3 is arbitrary and can be changed to any constant in the range $(1/2, 1)$.

Theorem C.3. *Every circuit class $\mathcal{C}$ is contained in its corresponding randomized circuit class. For example,* $\mathsf{AC}^0 \subseteq \mathsf{BPAC}^0$.

The other inclusion is not known in general. In the case of AC^0 and BPAC^0, restricting BPAC^0 to only $O(\log(n))$ random bits is enough to make it fall inside AC^0.

Theorem C.4. $\mathsf{BPAC}^0[\log(n)] \subseteq \mathsf{AC}^0$

Proof. In AC^0 we can enumerate all possible values for the random bits. Then, we compute the output of the randomized Boolean circuit for each case. There are $2^{O(\log n)} = \mathsf{poly}(n)$ possible random values. We can run the circuit for each of them in parallel, leading only to a polynomial blow-up in size but no blow-up in depth. Then, by definition of the randomized circuit, at least $2/3$ of the cases have the correct output. Since $2/3 > 1/2 + 1/7$, we can output $\mathrm{MAJ}_{1/7}$ of these results, which has an AC^0 circuit due to Theorem C.2. □

C.2 Boolean circuits and Turing machines

Most of the work in complexity theory is done over Turing machines. In this section, we present some known relations between circuits and Turing machines.

Definition C.15 (DTIME)**.** Let $T : \mathbb{N} \to \mathbb{N}$ be some function. A language $\mathcal{L}$ is in $\mathsf{DTIME}(T(n))$ iff there is a Turing machine that runs in time $c \cdot T(n)$ for some constant $c > 0$ and decides $\mathcal{L}$.

Definition C.16. $\mathsf{P} = \cup_{c \geq 0} \mathsf{DTIME}(n^c)$.

A probabilistic Turing machine is a Turing machine with access to random coins.

Definition C.17 (BPTIME)**.** Let $T : \mathbb{N} \to \mathbb{N}$ be some function. A language $\mathcal{L}$ is in $\mathsf{BPTIME}(T(n))$ iff there is a probabilistic Turing machine M that runs in time $c \cdot T(n)$ for some constant $c > 0$ and for all $x \in \{0,1\}^n$, $\Pr[M(x) = \mathcal{L}(x)] \geq 2/3$.

Definition C.18. $\mathsf{BPP} = \cup_{c \geq 0} \mathsf{BPTIME}(n^c)$.

When restricting $\mathsf{P/Poly}$ and $\mathsf{BPP/Poly}$ to their P-uniform families, they coincide with P and BPP, respectively. To simulate a circuit by a Turing machine, one first computes the circuit description (this can be done since we are only analyzing uniform families) and then evaluates it, both in polynomial time. For the simulation of Turing machines by circuits, the Cook–Levin construction works.

Theorem C.5. $\mathsf{Uniform} \cap \mathsf{P/Poly} = \mathsf{P}$

Theorem C.6. $\mathsf{Uniform} \cap \mathsf{BPP/Poly} = \mathsf{BPP}$

An important result relating machines with and without randomness is Adleman's theorem. Adleman (1978) proved that the non-uniformity of $\mathsf{P/Poly}$ gives enough power to simulate the randomness of BPP.

Theorem C.7 (Adleman's theorem)**.** $\mathsf{BPP} \subseteq \mathsf{P/Poly}$

The same argument applies directly to randomized circuits: after reducing the error probability of a polynomial-size randomized circuit C_n below 2^{-n} (by taking the majority of polynomially many independent copies), a union bound over all $x \in \{0,1\}^n$ shows that there is a fixed assignment r of the random gates that is correct for every input of length n. Hard-wiring r yields a deterministic circuit of polynomial size. Therefore:

Corollary C.1. $\mathsf{BPP/Poly} = \mathsf{P/Poly}$

D TRANSFORMER PRIMITIVES

In this appendix we introduce a set of primitive operations that are used throughout the formal proofs. The operations are applied as modifications to an already existing transformer. If the operations are applied to an entire family of transformers $\{\mathsf{TF}_n\}_{n\in\mathbb{N}}$ such that the language it computes is in $\mathsf{CoT}[T(n), d(n)]$, then the language of the modified family will be in the same class. This is because, although the operations add layers and coordinates to the embedding, they only add a constant number of layers and increase the embedding size by at most a constant factor.

Most operations do not make sense on their own, but they are used in later proofs.

It is interesting to remark that these are not only theoretical gadgets. It has been observed that transformers develop these types of structures during training, and that they can be identified by patterns in the attention matrices (Olsson et al., 2022).

D.1 FLAGGING TOKENS AND POSITIONS

For complex operations inside the loop of the transformer we will need to identify when a vector comes from a certain token or position. In this section we present how to mark the vectors with flags during the embedding.

For flagging a token we use the token embedding, and in the same way, for flagging a position we use the position encoding. Either way, the strategy is exactly the same. The embedding dimension is extended to allocate the flags. A new coordinate is added per flag. Therefore, using a constant number of flags only adds a constant to the embedding dimension. Thus, if the flagging is done in a family of transformers, its $\mathsf{CoT}[T(n), d(n)]$ class does not change.

For example, let v_σ and v_τ be vectors of a transformer TF where v_σ comes from embedding the token σ and v_τ from the token τ (with $\sigma \neq \tau$). We can define a transformer TF' that behaves like TF but recognizes when a vector comes from embedding the token σ. TF' has embedding dimension $d' = d + 1$ where d is the embedding dimension of TF.

$$v_\sigma = \begin{pmatrix} x \\ y \\ z \end{pmatrix} \to v'_\sigma = \begin{pmatrix} x \\ y \\ z \\ f_{\mathsf{on}} \end{pmatrix} \qquad v_\tau = \begin{pmatrix} i \\ j \\ k \end{pmatrix} \to v'_\tau = \begin{pmatrix} i \\ j \\ k \\ f_{\mathsf{off}} \end{pmatrix}$$

Figure 3: Change in a flagged and an unflagged vector.

This primitive also works as a mechanism to extend the embedding dimension without changing the behavior. If f_{on} and f_{off} are defined as 0, the resulting transformer is the same as the initial one with an extra coordinate that is always 0.

D.1.1 CONSTRUCTION FOR FLAGGING

Let TF be a transformer. We define TF' that behaves like TF but recognizes when a vector comes from embedding the token σ. TF' has embedding dimension $d' = d + 1$ where d is the embedding dimension of TF.

Essentially, the components of TF' are the same as those of TF. The main change is the token embedding $\theta'_{\mathsf{TE}} \colon \Sigma \to \mathbb{F}_{e,s}^{d+1}$, which is defined as

$$(\theta'_{\mathsf{TE}}(\tau))_j = \begin{cases} (\theta_{\mathsf{TE}}(\tau))_j & \text{if } j \neq d+1 \\ f_{\mathsf{on}} & \text{if } j = d+1 \text{ and } \tau = \sigma \\ f_{\mathsf{off}} & \text{if } j = d+1 \text{ and } \tau \neq \sigma \end{cases}$$

where f_{on} is the value that marks the vector as flagged and f_{off} as unflagged. These values can be chosen arbitrarily for each flag. In this case, the coordinate inserted for the flag is the last one, but it can be added anywhere.

The other components of TF' ignore the new $(d+1)$-th coordinate. The position encoding of TF' is equal to the position encoding of TF, except it puts a 0 in the $(d+1)$-th coordinate (not present in TF) independently of the position. The matrices of the ATTN, FF and OUTPUT layers of TF' are the ones of TF extended with zeros in the $(d+1)$-th row and column to match the change in embedding dimension.

TF' behaves exactly as TF except it has one more coordinate in the embedding. Said coordinate represents the flag and takes only the values f_{on} or f_{off} in every vector.

D.2 Preserve and ignore coordinates through ATTN and FF

When defining new transformers over existing ones we will need to extend the embedding dimension to store new information. In order to preserve the behavior of the original transformers across the attention and feed-forward layers, we need to construct a mechanism that ignores the extra coordinates. For example, this mechanism is used for preserving flags during ATTN and FF layers that do not use them.

Let TF be a transformer of embedding dimension d and assume we are constructing a transformer TF' of embedding dimension $d' = d+1$. Let W_Q, W_K, W_V, W_O be the matrices of an ATTN layer of TF and assume we want to perform the same layer in TF' but simulating that the ith coordinate does not exist. If we define the matrices of the layer W'_Q, W'_K, W'_V, W'_O as W_Q, W_K, W_V, W_O with a row and a column of zeros inserted at the ith position, we obtain the desired ATTN layer.

Note that because the ith row of W'_O is zero, the output vectors of the ATTN layer are 0 in the ith coordinate. This is what makes it possible not only to ignore the value of the ith coordinate, but also to leave it unaffected.

To preserve the coordinate without ignoring it, it suffices to set to zero only the corresponding row of W'_O. This construction works for more than one coordinate: to extend it to a set of coordinates, it suffices to set more rows and columns to 0.

The same applies to the feed-forward layer: making the ith column of W_1 zero makes it ignore the value of the ith coordinate, and making the ith row of W_2 and the ith entry of b_2 zero makes the output of the FF layer 0 in the ith coordinate, thus preserving its value.

$$\begin{pmatrix} A^{i\times i} & 0^{i\times 1} & B^{i\times(n-1-i)} \\ 0^{1\times i} & 0 & 0^{1\times(n-1-i)} \\ C^{(n-1-i)\times i} & 0^{(n-1-i)\times 1} & D^{(n-1-i)\times(n-1-i)} \end{pmatrix} \begin{pmatrix} v^{i\times 1} \\ x^{1\times 1} \\ w^{(n-1-i)\times 1} \end{pmatrix} = \begin{pmatrix} Av+Bw \\ 0 \\ Cv+Dw \end{pmatrix} \sim \begin{pmatrix} A & B \\ C & D \end{pmatrix} \begin{pmatrix} v \\ w \end{pmatrix}$$

Figure 4: Matrices for ignoring/preserving the $(i+1)$-th coordinate.

D.3 Add vector and constant functions

In this section, we develop a mechanism for adding a fixed vector to a previously flagged one. This technique works even when there is more than one flagged vector.

Assume there is a flag with values $f_{\mathsf{on}} = 1$ and $f_{\mathsf{off}} = 0$. For clarity, suppose the flag is in the last coordinate. Let v be the vector we want to add. We define a mechanism that replaces every h such that $h_d = f_{\mathsf{on}}$ with $h + v$.

The operation is performed with only one FF layer. We construct a layer such that $\mathsf{FF}(h) = \mathbb{I}(h_d = f_{\mathsf{on}}) \cdot v$. Said layer is achieved with the following parameters.

$$W_1 = id \qquad W_2 = \begin{pmatrix} 0 & \dots & 0 & v_1 \\ \vdots & \ddots & \vdots & \vdots \\ 0 & \dots & 0 & v_d \end{pmatrix} \qquad b_1 = b_2 = \begin{pmatrix} 0 \\ \vdots \\ 0 \end{pmatrix}$$

Notice that if the coordinate for the flag had been the k-th instead of the last one, then v would have appeared in the k-th column of W_2 instead of the last. If the flag must be kept for later layers, we take $v_d = 0$ (i.e., v is 0 in the flag coordinate).

This technique can be used to apply a constant function to the flagged vectors. This is achieved by exploiting the finite representation system. We can saturate each coordinate to B^+, then make it 0, and finally add our objective. This holds since for all $x, c \in \mathbb{F}_{e,s}$ it happens that $x + B^+ + B^+ + B^- + c = c$, because, as mentioned in Appendix B, $x + B^+ + B^+ = B^+$ and $B^+ + B^- = 0$.

The constant function $f(h) = v$ is implemented by adding four vectors, each with one FF layer as above. First, it is necessary to add the vector consisting entirely of B^+ twice; then, a vector consisting entirely of B^-; and finally, v; as shown in Equation 1. All these vectors have 0 in the flag coordinate, so that the flag is preserved across the four layers.

$$h \leftarrow h + \begin{pmatrix} B^+ \\ \vdots \\ B^+ \end{pmatrix} + \begin{pmatrix} B^+ \\ \vdots \\ B^+ \end{pmatrix} + \begin{pmatrix} B^- \\ \vdots \\ B^- \end{pmatrix} + v \tag{1}$$

D.4 Make a vector ignore others in an ATTN layer

We show how to make a vector ignore others in the ATTN layers, a technique that will prove useful in later constructions where we use one transformer to sequentially simulate several others. A vector h_i ignores vector h_j when the attention weight $(s_i)_j$ is 0, which can be achieved by making $\langle q_i, k_j \rangle = B^-$, since $\exp(B^-) = 0$.

We extend the embedding dimension with three more coordinates, while preserving the attention scores (previous to the $\mathrm{softmax}$) of the other vectors. We construct TF' based on TF such that the l-th vector ignores a set of vectors M in an ATTN layer; specifically, we require said layer to satisfy:

$$\langle q'_i, k'_j \rangle = \begin{cases} B^- & \text{if } i = l \wedge j \in M \\ \langle q_i, k_j \rangle & \text{if } i \neq l \vee j \notin M \end{cases}$$

This can be achieved by saturating the finite representation, since the attention score is $\langle q'_l, k'_j \rangle = \sum_{x=1}^{d+3} (q'_l)_x (k'_j)_x$. If the last two terms of the summation are B^-, then because of the iterative rounding, $\langle q'_l, k'_j \rangle = B^-$. Since we do not want to change the attention scores of vectors not in M, when $j \notin M$ the last three terms should be 0. The exact construction is presented in the following section.

D.4.1 Formal construction for ignoring vectors in an Attention layer

Let M be a set of indices. We construct an attention layer of TF' such that h_l ignores all the vectors h_j such that $j \in M$. As mentioned above, we require the attention scores (previous to the $\mathrm{softmax}$) to satisfy the following:

$$\langle q'_i, k'_j \rangle = \begin{cases} B^- & \text{if } i = l \wedge j \in M \\ \langle q_i, k_j \rangle & \text{if } i \neq l \vee j \notin M \end{cases}$$

We achieve this by saturating the finite representation, since $\langle q'_i, k'_j \rangle = \sum_{x=1}^{d+3} (q'_i)_x (k'_j)_x$.

We force $(k'_j)_{d+2}$ and $(k'_j)_{d+3}$ to be B^- when $j \in M$ (and 0 otherwise), and $(q'_i)_{d+2}$ and $(q'_i)_{d+3}$ to be 1 when $i = l$ and 0 otherwise. We also make $(k'_j)_{d+1}$ and $(q'_i)_{d+1}$ equal to 0 for all i, j, so the third to last term of the inner product is always 0. With these values, the only attention scores modified are the ones of the lth vector, since every other vector has 0 in the last three coordinates of its query, thus making the last three terms of the inner product 0 and therefore not affecting the score.

This can be achieved with the following flags:

$$(h_i)_{d+1} = \begin{cases} 1 & \text{if } i = l \\ 0 & \text{otherwise} \end{cases} \qquad (h_i)_{d+2} = (h_i)_{d+3} = \begin{cases} 1 & \text{if } i \in M \\ 0 & \text{otherwise} \end{cases}$$

Now the query and key matrices are extended as:

$$W'_Q = \begin{pmatrix} W_Q & 0^{\mathbb{F}_{e,s}^{d\times 1}} & 0^{\mathbb{F}_{e,s}^{d\times 1}} & 0^{\mathbb{F}_{e,s}^{d\times 1}} \\ 0^{\mathbb{F}_{e,s}^{1\times d}} & 0 & 0 & 0 \\ 0^{\mathbb{F}_{e,s}^{1\times d}} & 1 & 0 & 0 \\ 0^{\mathbb{F}_{e,s}^{1\times d}} & 1 & 0 & 0 \end{pmatrix} \qquad W'_K = \begin{pmatrix} W_K & 0^{\mathbb{F}_{e,s}^{d\times 1}} & 0^{\mathbb{F}_{e,s}^{d\times 1}} & 0^{\mathbb{F}_{e,s}^{d\times 1}} \\ 0^{\mathbb{F}_{e,s}^{1\times d}} & 0 & 0 & 0 \\ 0^{\mathbb{F}_{e,s}^{1\times d}} & 0 & B^- & 0 \\ 0^{\mathbb{F}_{e,s}^{1\times d}} & 0 & 0 & B^- \end{pmatrix}$$

Then,

$$\begin{aligned}
\langle q'_i, k'_j \rangle &= \sum_{x=1}^{d+3} (q'_i)_x (k'_j)_x = \langle q_i, k_j \rangle + (q'_i)_{d+1}(k'_j)_{d+1} + (q'_i)_{d+2}(k'_j)_{d+2} + (q'_i)_{d+3}(k'_j)_{d+3} \\
&= \langle q_i, k_j \rangle + 0 + (q'_i)_{d+2}(k'_j)_{d+2} + (q'_i)_{d+3}(k'_j)_{d+3} \\
&= \langle q_i, k_j \rangle + (q'_i)_{d+2}(k'_j)_{d+2} + (q'_i)_{d+3}(k'_j)_{d+3} \\
&= \begin{cases} \langle q_i, k_j \rangle + 1 \cdot B^- + 1 \cdot B^- & \text{if } i = l \wedge j \in M \\ \langle q_i, k_j \rangle + 1 \cdot 0 + 1 \cdot 0 & \text{if } i = l \wedge j \notin M \\ \langle q_i, k_j \rangle + 0 \cdot B^- + 0 \cdot B^- & \text{if } i \neq l \wedge j \in M \\ \langle q_i, k_j \rangle + 0 \cdot 0 + 0 \cdot 0 & \text{if } i \neq l \wedge j \notin M \end{cases} \\
&= \begin{cases} \langle q_i, k_j \rangle + B^- + B^- & \text{if } i = l \wedge j \in M \\ \langle q_i, k_j \rangle + 0 + 0 & \text{if } i = l \wedge j \notin M \\ \langle q_i, k_j \rangle + 0 + 0 & \text{if } i \neq l \wedge j \in M \\ \langle q_i, k_j \rangle + 0 + 0 & \text{if } i \neq l \wedge j \notin M \end{cases} \\
&= \begin{cases} B^- & \text{if } i = l \wedge j \in M \\ \langle q_i, k_j \rangle & \text{otherwise} \end{cases}
\end{aligned}$$

Namely, $\langle q'_i, k'_j \rangle = B^-$ if $i = l$ and $j \in M$, but $\langle q'_i, k'_j \rangle = \langle q_i, k_j \rangle$ otherwise.

D.5 Linear transformations

Although it may seem that transformers should be able to perform linear transformations natively, this is not the case. Applying a linear transformation to a vector is not straightforward, since in every ATTN and FF layer the result of the layer is added to the original vector instead of replacing it. An intuitive idea would be to apply the transformation $M - I$ when we want to transform the vector x to Mx, since $x + (M - I)x = x + Mx - Ix = Mx$. This does not hold in our model, since addition and multiplication in $\mathbb{F}_{e,s}$ are not associative (nor distributive), so rounding errors may appear. To avoid this representation error, we extend the embedding dimension to get more memory per vector, so that we can store the value of Mx in new coordinates and then move it to the original ones.

We give a construction that applies the linear transformation M to a previously flagged vector. Let h_f be the flagged vector. As mentioned, we need to double the embedding dimension (from d to $2d$) so we can have coordinates to use as temporary memory. Our construction works with two ATTN layers operating as shown in Figure 5 (see Appendix D.5.1 for the exact construction). The first ATTN layer computes the transformation, saving its result in the coordinates $d + 1$ to $2d$. Then, the second layer moves the result to the right place.

D.5.1 Construction for Linear transformations

We show a mechanism to replace a flagged vector with the result of applying the linear transformation M to it. Let f denote the index of the flagged vector. We perform the operation with two consecutive ATTN layers, in which h_f only attends to itself. Therefore, after these two layers h_f will be replaced with Mh_f.

$$\widetilde{h_f} = \begin{pmatrix} (h_f)_1 \\ \vdots \\ (h_f)_d \\ 0 \\ \vdots \\ 0 \end{pmatrix} \xrightarrow{\text{first ATTN layer}} \widetilde{h_f} = \begin{pmatrix} 0 \\ \vdots \\ 0 \\ (Mh_f)_1 \\ \vdots \\ (Mh_f)_d \end{pmatrix} \xrightarrow{\text{second ATTN layer}} \widetilde{h_f} = \begin{pmatrix} (Mh_f)_1 \\ \vdots \\ (Mh_f)_d \\ 0 \\ \vdots \\ 0 \end{pmatrix}$$

Figure 5: Visual steps for linear transformations.

We need to double the embedding size from d to $2d$ (plus some flags) since we need to store the result of the transformation in some coordinates (the added $d+1$ to $2d$) as an intermediate step. In the first ATTN layer, we compute the transformation, saving its result in the coordinates $d+1$ to $2d$. Then, in the second layer we move the result to the right place.

$$\widetilde{h_f} = \begin{pmatrix} h_f \\ 0^{d\times 1} \end{pmatrix} \xrightarrow{\text{first ATTN layer}} \widetilde{h_f} = \begin{pmatrix} 0^{d\times 1} \\ Mh_f \end{pmatrix} \xrightarrow{\text{second ATTN layer}} \widetilde{h_f} = \begin{pmatrix} Mh_f \\ 0^{d\times 1} \end{pmatrix}$$

We assume that the coordinates $d+1$ to $2d$ (used as memory) are set to 0 by the layers preceding this mechanism. This can be achieved by keeping them at 0 from the beginning, or by setting them to 0 with the primitive described in Appendix D.3.

Let us construct each of the ATTN layers. As mentioned before, in both layers we need $\widetilde{h_f}$ to only attend to itself. Therefore, we define W_Q and W_K using the strategy of Appendix D.4.

Recall from the definition of the ATTN mechanism (Algorithm 2) that the output of the ATTN layer is $h'_i = W_O \sum_{j=1}^{a} (s_i)_j v_j$. Since $\widetilde{h_f}$ only attends to itself, we have

$$h'_f = W_O \sum_{j=1}^{a} (s_f)_j v_j = W_O v_f = W_O(W_V \widetilde{h_f}).$$

- For the first layer, defining W_O and W_V as follows

$$W_V = id \qquad\qquad W_O = \begin{pmatrix} -id^{\mathbb{R}^{d\times d}} & 0^{\mathbb{R}^{d\times d}} \\ M & 0^{\mathbb{R}^{d\times d}} \end{pmatrix}$$

  yields $h'_f = (-(h_f)_1, \ldots, -(h_f)_d, (Mh_f)_1, \ldots, (Mh_f)_d)$, which, added to $\widetilde{h_f}$, gives us what we were looking for after the first layer.

- For the second ATTN layer, we use the same mechanism for making $\widetilde{h_f}$ only attend to itself. Defining W_V and W_O as follows

$$W_V = id \qquad\qquad W_O = \begin{pmatrix} 0^{\mathbb{R}^{d\times d}} & id^{\mathbb{R}^{d\times d}} \\ 0^{\mathbb{R}^{d\times d}} & -id^{\mathbb{R}^{d\times d}} \end{pmatrix}$$

  we get $h'_f = ((Mh_f)_1, \ldots, (Mh_f)_d, -(Mh_f)_1, \ldots, -(Mh_f)_d)$, which, added to $\widetilde{h_f}$, gives us what we were looking for after the second layer.

Note that, after these two layers, the values of h_i for all $i \neq f$ have changed. Yet, this is not relevant with respect to the way we use the primitive.

D.6 MAX COORDINATE

We show how to take the maximum of a set of coordinates. This operation will be needed for later constructions.

We give a construction for taking the maximum of two coordinates. This construction can be easily extended to more coordinates. The operation works properly when the set of coordinates has a unique maximum. All the coordinates that are not the maximum are replaced by 0.

$$\begin{pmatrix} a \\ b \\ \vdots \end{pmatrix} \to \begin{cases} \begin{pmatrix} a \\ 0 \\ \vdots \end{pmatrix} & \text{if } a > b \\ \\ \begin{pmatrix} 0 \\ b \\ \vdots \end{pmatrix} & \text{if } b > a \end{cases}$$

The maximum of two coordinates can be computed with the following primitives:

$$\begin{pmatrix} a \\ b \\ 0 \\ 0 \\ \vdots \end{pmatrix} \xrightarrow{\text{FF layer}} \begin{pmatrix} a \\ b \\ \text{relu}(a-b) \\ \text{relu}(b-a) \\ \vdots \end{pmatrix} \xrightarrow[\text{the coordinates with the relu}]{\text{Multiply twice by } B^+} \begin{pmatrix} a \\ b \\ B^+ \cdot \mathbb{I}(a > b) \\ B^+ \cdot \mathbb{I}(b > a) \\ \vdots \end{pmatrix} \longrightarrow$$

$$\xrightarrow[\text{coordinates with the comparison}]{\text{Divide by } B^+ \text{ the}} \begin{pmatrix} a \\ b \\ \mathbb{I}(a > b) \\ \mathbb{I}(b > a) \\ \vdots \end{pmatrix} \xrightarrow[\text{operation}]{\text{Special}} \begin{pmatrix} a \cdot \mathbb{I}(a > b) \\ b \cdot \mathbb{I}(b > a) \\ 0 \\ 0 \\ \vdots \end{pmatrix} = \begin{cases} \begin{pmatrix} a \\ 0 \\ \vdots \end{pmatrix} & \text{if } a > b \\ \\ \begin{pmatrix} 0 \\ b \\ \vdots \end{pmatrix} & \text{if } b > a \end{cases}$$

For computing the maximum of a set of coordinates, it suffices to compute a comparison for each pair of the set and multiply each coordinate x by $\mathbb{I}(x > y)$ for all $y \neq x$ in the set.

D.6.1 Implementation of the special operation and first step of maximum algorithm

The FF layer needed for the first step transforms $(a, b, 0, 0)$ into $(a, b, \text{relu}(a-b), \text{relu}(b-a))$. It consumes $(a, b, 0, 0)$ and outputs $(0, 0, \text{relu}(a-b), \text{relu}(b-a))$, which, added to $(a, b, 0, 0)$, gives what we need. Instantiating the matrices as follows produces the desired effect:

$$W_1 = \begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 1 & -1 & 0 & 0 \\ -1 & 1 & 0 & 0 \end{pmatrix} \qquad W_2 = \begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \qquad b_1 = b_2 = \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \end{pmatrix}$$

The special operation is more complicated. It is not a linear transformation; therefore, we need to define a new technique. Recall the definitions of $\mathbb{I}_a$ and $\mathbb{I}_b$. $\mathbb{I}_a$ is obtained from $\text{relu}(a-b)$ by the saturation steps above. It is the indicator function of $a > b$; therefore, $\mathbb{I}_a$ equals 1 if $a > b$ and 0 otherwise. Notice that it cannot occur that $a \neq b$ but the rounded value of $a - b$ is 0, since the numbers near 0 are more densely represented. $\mathbb{I}_b$ indicates $b > a$ in the same way as $\mathbb{I}_a$. Observe that one of $\mathbb{I}_a$ and $\mathbb{I}_b$ equals 1 and the other equals 0. Then, $\mathbb{I}_b = 1 - \mathbb{I}_a$ and $\mathbb{I}_a = 1 - \mathbb{I}_b$.

$$\begin{pmatrix} a \\ b \\ \mathbb{I}_a \\ \mathbb{I}_b \end{pmatrix} \xrightarrow[\text{operation}]{\text{Special}} \begin{pmatrix} a \cdot \mathbb{I}_a \\ b \cdot \mathbb{I}_b \\ 0 \\ 0 \end{pmatrix}$$

This operation is performed with three FF layers. Naturally, after each layer a new vector is added to the original. The vectors we add are the following:

$$\begin{pmatrix} a \\ b \\ \mathbb{I}_a \\ \mathbb{I}_b \end{pmatrix} \xrightarrow[\text{operation}]{\text{Special}} \begin{pmatrix} a & + & B^+ \cdot \mathbb{I}_b & + & B^+ \cdot \mathbb{I}_b & + & B^- \cdot \mathbb{I}_b \\ b & + & B^+ \cdot \mathbb{I}_a & + & B^+ \cdot \mathbb{I}_a & + & B^- \cdot \mathbb{I}_a \\ \mathbb{I}_a & + & 0 & + & 0 & + & -\mathbb{I}_a \\ \mathbb{I}_b & + & 0 & + & 0 & + & -\mathbb{I}_b \end{pmatrix}$$

Note that if $\mathbb{I}_a = 1$, then $\mathbb{I}_b = 0$, so the first coordinate is not modified and the second is changed to 0 because of the finite representation properties. In the same way, if $\mathbb{I}_a = 0$, then $\mathbb{I}_b = 1$, so the second coordinate is not modified and the first is changed to 0.

Let us show how to instantiate the FF layers. The first two have parameters:

$$W_1 = id \qquad W_2 = \begin{pmatrix} 0 & 0 & 0 & B^+ \\ 0 & 0 & B^+ & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix} \qquad b_1 = b_2 = \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \end{pmatrix}$$

Notice that the output of these FF layers when the input is $(x, y, \mathbb{I}_a, \mathbb{I}_b)$ will be $(B^+ \cdot \mathbb{I}_b, B^+ \cdot \mathbb{I}_a, 0, 0)$.

The third FF layer has parameters:

$$W_1 = id \qquad W_2 = \begin{pmatrix} 0 & 0 & 0 & B^- \\ 0 & 0 & B^- & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & -1 \end{pmatrix} \qquad b_1 = b_2 = \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \end{pmatrix}$$

which, when the input is $(x, y, \mathbb{I}_a, \mathbb{I}_b)$, outputs $(B^- \cdot \mathbb{I}_b, B^- \cdot \mathbb{I}_a, -\mathbb{I}_a, -\mathbb{I}_b)$.

D.7 Repeat a token once it is printed

We show how to make a transformer repeat a token until we decide to make it stop. This construction is needed to argue that for every transformer TF there is a transformer TF' of the same $\mathsf{CoT}[T(n), d(n)]$ class that computes the same language and consumes all of its budget. Intuitively, we flag the token σ to be repeated and apply a constant function at the end of the body of the transformer to all the flagged vectors. The constant function forces the probability distribution to have all the probability concentrated in the token σ.

Assume that TF is normalized and let us define TF'. TF' is identical to TF except for the following modifications. Let σ be the token to repeat. We flag the vectors coming from σ with the token embedding. Since we are replicating a token that was printed, i.e., it is not part of the input, its corresponding vector is the last one. This is because we begin repeating it after the first time it is printed.

Since we flag the vectors coming from the token σ, we can apply a constant function only to them. This function is applied in the last layer of the body of TF'. Since we assume TF was normalized, the OUTPUT matrix is just a projection. Then, the value of the constant function is what is output as the probability distribution (after $\mathrm{softmax}$). If the constant function has value B^- in every coordinate except in the one associated with the probability of σ, where it has B^+, the distribution output by the distribution generator will have all the probability concentrated in σ. Thus, TF' prints σ.

Observe that this only works if the token does not appear in the input word. The case where it does appear can be handled by exploiting that for each input size we have a different position encoding.

Since the input word has size n, we can use the PE to flag the first n vectors. Then, we can use that flag to set to 0 the coordinate used by the TE for flagging the token to be repeated, i.e., turning off the flag. We make them 0 using a constant function as described in Section D.3.

D.8 Force to halt after a certain number of steps of CoT

Using a strategy similar to the one in Section D.7, we use the PE to flag when a certain position is reached. Then, if the transformer is normalized, by modifying the last vector we decide the output distribution. In particular, we can force it to concentrate the probability in q_{accept} or q_{reject}. We change the last vector with a constant function. The flag inserted by the PE is used to decide whether the function has to be applied or not.

Notice that this primitive, together with the ability to repeat a token, shows that for every transformer TF in $\mathsf{CoT}[T(n), d(n)]$ there is a transformer TF' in the same class that always consumes all of its budget.

We extend the alphabet with two dummy versions of the decision tokens and construct TF' based on TF. We make TF' work as TF, but we make it print the dummy versions of the decision tokens when TF would have printed the real ones[8]. Then, we force TF' to repeat the dummy decision tokens until it reaches the last position of the tape (i.e., uses all the budget). At that point, we force TF' to print the real decision token corresponding to the one it was repeating.

D.9 Copy one vector into another with ATTN

For the disjunction of transformers we will need to move values from the ith vector to the jth vector. In this section we show how to do it. This operation can only be done if $i < j$, since the attention mechanism only allows a vector to attend to vectors to its left, i.e., h_k can attend to h_l if and only if $l \leq k$.

Let j be the index of the receiving vector and i the index of the copied one, i.e., we copy from h_i to h_j. The operation is done in two steps.

First, we set to 0 the coordinates of h_j that will receive the values. This is done with a constant function.

Second, we make h_j only attend to h_i in an attention layer. In said attention layer, we use $W_V = id$ and W_O such that it outputs the desired coordinates of h_i in the target coordinates. Formally, the k-th row of W_O is zero if h_j is not receiving anything from h_i in the k-th coordinate. If h_j is receiving the coordinate l of h_i in the position k, then the kth row of W_O has 0 in every position except the lth, where it has 1.

For example, if we want to copy the 2nd coordinate of h_i to the 3rd coordinate of h_j, W_O would look like Equation equation 2. Then, the vector coming out of the ATTN layer will have 0 in every coordinate, except in the 3rd, where it will be $(h_i)_2$.

$$W_O = \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix} \tag{2}$$

[8] This is achieved by giving the probability of each decision token to its dummy version.

E NORMAL FORM OF DETERMINISTIC TRANSFORMERS

To simplify the handling of transformers, we define a notion of normalized deterministic transformer. This notion specifies properties of the OUTPUT matrix and of the vector it receives as input.

Definition E.1 (Normalized deterministic transformer)**.** We say a deterministic transformer is normalized if it satisfies two conditions.

- N1: The matrix of the OUTPUT layer has to be a projection onto the first $|\Sigma|$ coordinates, i.e., $\theta_{\mathsf{OUTPUT}} = \left(id^{|\Sigma|\times|\Sigma|} \;\; 0^{|\Sigma|\times(d-|\Sigma|)}\right)$.
- N2: The last vector after the last layer must consist entirely of B^- in every projected coordinate except one, which must equal B^+.

The motivation for this definition is to simplify the OUTPUT layer (N1) and to have more control over the probability distribution generated (N2). The exact values of the last vector are such that after projecting and applying $\mathrm{softmax}$ in the OUTPUT layer we get a deterministic distribution with all the probability concentrated in only one token.

For every family of transformers $\{\mathsf{TF}_n\}_{n\in\mathbb{N}}$ in $\mathsf{CoT}[T(n), d(n)]$ computing the language $\mathcal{L}$, there is a family $\{\mathsf{TF}'_n\}_{n\in\mathbb{N}}$ in the same class, computing the same language, where for all n, TF'_n is normalized. We give an algorithm for constructing the normalized counterpart of a given transformer. First we make it meet N1 and then N2.

E.1 ALGORITHM FOR N1

Notice that every linear transformation $M : \mathbb{F}^d_{e,s} \to \mathbb{F}^{|\Sigma|}_{e,s}$ with $d \geq |\Sigma|$ can be decomposed into two steps.[9]

$$\begin{pmatrix} x_1 \\ \vdots \\ x_d \end{pmatrix} \xrightarrow{M'} \begin{pmatrix} (Mx)_1 \\ \vdots \\ (Mx)_{|\Sigma|} \\ v \end{pmatrix} \xrightarrow{\text{projection}} \begin{pmatrix} (Mx)_1 \\ \vdots \\ (Mx)_{|\Sigma|} \end{pmatrix}$$

where $v \in \mathbb{F}^{d-|\Sigma|}_{e,s}$ can have any values.

Let TF be a transformer and M be the matrix of the OUTPUT layer. Using the decomposition just presented, and the strategy defined in Appendix D.5 for applying linear transformations, we can change the matrix of the OUTPUT layer to be only a projection. The linear transformation M' has to be applied to the last vector, since it is the one going from the body of TF to the OUTPUT layer.

E.2 ALGORITHM FOR N2

Let TF be a transformer satisfying N1. We give an algorithm for making it also meet N2.

We use the maximum operation described in Appendix D.6 to make all the first $|\Sigma|$ coordinates of the last vector 0, except the one that was the maximum before. Then, if the maximum is positive, multiplying twice by B^+, then subtracting 1 from every coordinate and multiplying by B^+ again makes TF meet N2.

[9] We can always add more coordinates to the embedding if $d < |\Sigma|$. Since the alphabet Σ is fixed, this adds only a constant number of coordinates, thus not changing the class of the transformer.

$$\begin{pmatrix} 0 \\ \vdots \\ 0 \\ x_{max} \\ 0 \\ \vdots \\ 0 \end{pmatrix} \xrightarrow{\text{multiplying twice by } B^+} \begin{pmatrix} 0 \\ \vdots \\ 0 \\ B^+ \\ 0 \\ \vdots \\ 0 \end{pmatrix} \xrightarrow{\text{subtracting 1}} \begin{pmatrix} -1 \\ \vdots \\ -1 \\ B^+ - 1 \\ -1 \\ \vdots \\ -1 \end{pmatrix} \xrightarrow[B^+]{\text{multiplying by}} \begin{pmatrix} B^- \\ \vdots \\ B^- \\ B^+ \\ B^- \\ \vdots \\ B^- \end{pmatrix}$$

It could be the case that the maximum was negative; in that case the resulting vector is not the one we presented. We show a mechanism for making the maximum strictly positive, so that we can insert it in the algorithm before the multiplications by B^+.

At this point our vector looks like $(0, \ldots, 0, x_{max}, 0, \ldots, 0)$; we transform it into $(0, \ldots, 0, |x_{max}|, 0, \ldots, 0)$. Recall that the coordinates we are interested in are only the first $|\Sigma|$. Let v denote the block containing these coordinates. Since $\mathrm{relu}(x) + \mathrm{relu}(-x) = |x|$ for all x, our algorithm does the following:

$$\begin{pmatrix} v \\ \vdots \end{pmatrix} \to \begin{pmatrix} v \\ -v \\ \vdots \end{pmatrix} \to \begin{pmatrix} \mathrm{relu}(v) \\ \mathrm{relu}(-v) \\ \vdots \end{pmatrix} \to \begin{pmatrix} \mathrm{relu}(v) + \mathrm{relu}(-v) \\ \vdots \end{pmatrix} = \begin{pmatrix} |v| \\ \vdots \end{pmatrix}$$

where relu is applied coordinate-wise.

The first step is done in the same way as the first half of the linear transformations (Appendix D.5), writing $-v$ in fresh coordinates. For the second step, notice that $\mathrm{relu}(x)$ can be computed with one FF layer ($W_2 = W_1 = id$, $b_2 = b_1 = 0$). Thus, $\mathrm{relu}(v)$ and $\mathrm{relu}(-v)$ can be computed with one FF layer and stored in fresh coordinates. Next, they are moved to the desired positions. Finally, for adding both vectors we can use one FF layer with the following parameters. Observe that this works since all the coordinates of $\mathrm{relu}(-v)$ are non-negative; thus, $\mathrm{relu}(\mathrm{relu}(-v)) = \mathrm{relu}(-v)$.

$$W_1 = id^{d \times d} \quad W_2 = \begin{pmatrix} 0^{|\Sigma| \times |\Sigma|} & id^{|\Sigma| \times |\Sigma|} & 0^{|\Sigma| \times (d-2|\Sigma|)} \\ 0^{(d-|\Sigma|) \times |\Sigma|} & 0^{(d-|\Sigma|) \times |\Sigma|} & 0^{(d-|\Sigma|) \times (d-2|\Sigma|)} \end{pmatrix} \quad b_1 = b_2 = 0^{d \times 1}$$

F PROOFS OF CoT AND BPCoT CLOSURE PROPERTIES

F.1 COMPLEMENT

In this section we prove Proposition 3.1 for $\mathsf{CoT}[T(n), d(n)]$. The proof for BPCoT is the same.

Let TF be a transformer, we construct TF' that flips the decision token of TF, i.e., if $\mathsf{TF}^*(\alpha) = q_{accept}$, then $\mathsf{TF}'^*(\alpha) = q_{reject}$ and if $\mathsf{TF}^*(\alpha) = q_{reject}$, then $\mathsf{TF}'^*(\alpha) = q_{accept}$. This is the only change made to TF, thus $\mathsf{TF}'^i(\sigma) = \mathsf{TF}^i(\sigma)$ for every i when σ is not a decision symbol.

This is done by swapping the rows associated with q_{accept} and q_{reject} in the matrix of the OUTPUT layer of TF. Thus, when the largest probability is associated with q_{accept} in TF, that same number will be associated with q_{reject} in TF'.

Formally, TF' is defined exactly the same as TF except in the OUTPUT layer, where the matrix is different. Let M be the matrix of the OUTPUT layer of TF, let i be the position of the token q_{accept} and j the position of q_{reject}, then the matrix of the OUTPUT layer of TF' will be $M' = PM$ where P is the matrix that flips the rows i and j.

$$M'_{kl} = \begin{cases} 1 & \text{if } k = l,\ k \neq i \text{ and } k \neq j \\ 1 & \text{if } k = j \text{ and } l = i \\ 1 & \text{if } k = i \text{ and } l = j \\ 0 & \text{otherwise} \end{cases}$$

Since the OUTPUT matrix is the only difference between TF and TF', they have the same budget and embedding size as TF, therefore, TF' belongs to the same class as TF.

We can define the family $\{\mathsf{TF}'_n\}_{n\in\mathbb{N}}$ where TF'_i is the negation of TF_i constructed as explained in this section. Then $\{\mathsf{TF}'_n\}_{n\in\mathbb{N}}$ computes the complement of $\{\mathsf{TF}_n\}_{n\in\mathbb{N}}$. Since for every $i \in \mathbb{N}$, TF'_i has the same budget and embedding size as TF_i, $\bar{\mathcal{L}}$ is in $\mathsf{CoT}[T(n), d(n)]$.

F.2 DISJUNCTION

In this section we give the construction for Proposition 3.2. We give the construction for deterministic transformers. It works in the probabilistic setting if the error is reduced to at most $1/5$, which Proposition 3.3 allow us to assume. The error needs to be reduced to $1/5$ so the disjunction of one run of each transformer has probability larger than $1/2$.

Let $\mathcal{L}_1$ and $\mathcal{L}_2$ be languages in $\mathsf{CoT}[T(n), d(n)]$, then there is a family of transformers $\{(\mathsf{TF}_1)_n\}_{n\in\mathbb{N}}$ (resp. $\{(\mathsf{TF}_2)_n\}_{n\in\mathbb{N}}$) of embedding size $d_1(n)$ (resp. $d_2(n)$) $\in O(d(n))$ that after $T_1(n)$ (resp. $T_2(n)$) $\in T(n)$ steps of CoT computes $\mathcal{L}_1$ (resp. $\mathcal{L}_2$).

In this appendix, we construct a family $\{\mathsf{TF}_n\}_{n\in\mathbb{N}}$ that computes $\mathcal{L}_1 \cup \mathcal{L}_2$. It does it in $T_1(n) + T_2(n) + 1 \in O(T(n))$ steps of CoT and it has embedding size $O(d(n))$.

As mentioned in the main text and illustrated in Figure 6, TF first runs TF_1, then TF_2, and lastly, computes the disjunction of the outputs.

First, observe that for every input α, the tape of a transformer, at the end of the computation, ends up looking as $\alpha\beta q$, where q is the decision token printed by the transformer and β ($\in \Sigma^*$) is the sequence of intermediate tokens printed during the chain of thought. We denote by $\alpha\beta q_1$ the content of the tape of TF_1 at the end of the computation when the input word is α. Similarly, we denote by $\alpha\gamma q_2$ the content of the tape of TF_2 at the end of the computation when the input word is α. The tape of TF, computing the disjunction of TF_1 and TF_2, at the end of the computation will have content $\alpha\beta q_1\gamma q_2 q$, where q is the decision token representing the disjunction of q_1 and q_2.

Let us construct TF from TF_1 and TF_2. Since we cannot change the body of the transformer during the iterations, we are required to use some tricks.

First, we grow the dimension of the embedding for running the body of both transformers at the same time over the same vectors but independently. In the final layers we choose which probability

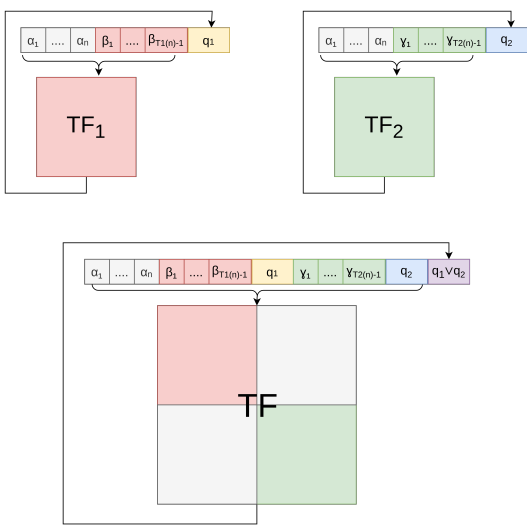

Figure 6: Visual representation of the construction for disjunction

distribution will go to the output layer, this makes TF print the token generated by TF_1 or TF_2. As always, we assume that the transformers are normalized.

We use the first half of the embedding for the transformer TF_1 and the second for the transformer TF_2. We have some extra coordinates for flags at the end, but we will ignore them for now.

$$\mathsf{TE}(\sigma)_i = \begin{cases} \mathsf{TE}_1(\sigma)_i & \text{if } i \leq d_1(|\alpha|) \\ \mathsf{TE}_2(\sigma)_{i-d_1(|\alpha|)} & \text{if } d_1(|\alpha|) < i \leq d_1(|\alpha|) + d_2(|\alpha|) \\ 0 & \text{if } d_1(|\alpha|) + d_2(|\alpha|) < i \end{cases} \tag{3}$$

$$\mathsf{PE}(n)_i = \begin{cases} \mathsf{PE}_1(n)_i & \text{if } i \leq d_1(|\alpha|) \\ \text{shifted}\mathsf{PE}_2(n)_{i-d_1(|\alpha|)}{}^{10} & \text{if } d_1(|\alpha|) < i \leq d_1(|\alpha|) + d_2(|\alpha|) \\ \text{flags} & \text{if } d_1(|\alpha|) + d_2(|\alpha|) < i \end{cases} \tag{4}$$

The transformer TF has one layer for each layer of TF_1 and each of TF_2. The first layers correspond to the body of TF_1. They are constructed ignoring the half of the embedding storing the coordinates used by TF_2. The second half of layers are the other way around, they compute the layers of TF_2 and ignore the TF_1 coordinates. Then, all the matrices in the layers of the first part of the body of TF look like V_1 and the ones in the second layers will look like V_2 where W_1 and W_2 are the matrices of the corresponding layers in TF_1 and TF_2.

$$V_1 = \begin{pmatrix} W_1 & 0 \\ 0 & 0 \end{pmatrix} \qquad\qquad V_2 = \begin{pmatrix} 0 & 0 \\ 0 & W_2 \end{pmatrix}$$

It is important to notice that the values of the coordinates corresponding to one transformer do not affect the values of the other. Thus, if the tape of the transformer has content γ, then at the end of the body of TF, the last vector would look like (x, y, flags) where x is the last vector at the end of the body of TF_1 when its tape content is γ, and y the one at the end of TF_2 when its tape content is γ. Therefore, if we always choose to transform said vector into $(x, 0, \ldots, 0)$, then TF will produce the same tape as TF_1. This works for generating the first $T_1(|\alpha|)$ tokens which are βq_1. Thus, we would like to apply this transformation only to the vectors generating them. Therefore, we need to flag them with the position encoding. Since $h_{|\alpha|+i}$ carries the distribution to output in the

[10] Defined later in the construction.

$(i+1)$-th step, i.e., the $(|\alpha|+i+1)$-th token of the tape, the vectors that we need to flag will be $h_{|\alpha|}, h_{|\alpha|+1}, \ldots, h_{|\alpha|+T_1(|\alpha|)-1}$.

After running $T_1(|\alpha|)$ steps of this construction, the tape will be $\alpha\beta q_1$. At this point, we want to start generating γ, i.e., the tape of TF_2. Just changing what we choose from $(x,0,0)$ to $(y,0,0)$ before the OUTPUT layer does not work, since what the construction will produce would be $\mathsf{TF}_2(\alpha\beta q_1)$, and we need $\mathsf{TF}_2(\alpha)$.

Now we show how to run TF_2 in this construction ignoring the tokens produced by TF_1. We make TF_2 not attend to the tokens in positions $|\alpha|+1$ to $T_1(|\alpha|)$ (this can be done using the primitive presented in the Section D.4). We also need for them to be invisible to the portion of the position encoding corresponding to TF_2. Thus, we define the shifted version of the position encoding mentioned in Equation F.2 as follows.

$$\mathsf{PE}(n)_i = \begin{cases} \mathsf{PE}_1(n)_i & \text{if } i \leq d_1(|\alpha|) \\ \text{shifted}\mathsf{PE}_2(n)_{i-d_1(|\alpha|)} & \text{if } d_1(|\alpha|) < i \leq d_1(|\alpha|)+d_2(|\alpha|) \\ \text{flags} & \text{if } d_1(|\alpha|)+d_2(|\alpha|) < i \end{cases}$$

$$\text{shifted}\mathsf{PE}_2(n) = \begin{cases} \mathsf{PE}_2(n) & \text{if } n \leq |\alpha| \\ 0 & \text{if } |\alpha| < n < T_1(|\alpha|) \\ \mathsf{PE}_2(n-T_1(|\alpha|)) & \text{if } T_1(|\alpha|) < n \end{cases}$$

The first case is for the positions occupied by the input, we do not want to change their encoding. The second case is for the positions occupied by the tokens generated by TF_1, we do not care about them. The third case is for the tokens that TF_2 will generate. It should believe that they are in their natural positions, i.e., shifted $T_1(|\alpha|)$ positions left (the positions occupied by the tokens generated by TF_1). We also need to change the flag used for choosing the distribution in the vectors at positions after $|\alpha|+T_1(|\alpha|)$ so TF chooses the distribution corresponding to TF_2 instead of TF_1.

At this point, the construction almost produces the tape of TF_1 followed by the tape of TF_2 except for one token. We need to do something special for the first token generated by TF_2. The distribution corresponding to it is in the vector $h_{|\alpha|}$ but the last vector of the first run computing TF_2 tokens is $h_{|\alpha|+T_1(|\alpha|)}$, the one coming from the last token of TF_1. Therefore, before choosing the distribution, we need to copy the vector $h_{|\alpha|}$ to $h_{|\alpha|+T_1(|\alpha|)}$. Notice that nothing more is needed, since the attention mechanism only attends to vectors to the right, so $h_{|\alpha|}$ is not affected by the vectors coming from the tokens produced by TF_1. This is done with the operation presented in Section D.9.

With this last fix, TF generates the tape of TF_1 followed by the tape of TF_2. We now show how to compute q, i.e., the disjunction of q_1 and q_2. This is easily done making $h_{|\alpha|+T_1(|\alpha|)+T_2(\alpha)}$ only attend to itself and to $h_{|\alpha|+T_1(|\alpha|)}$, i.e., to the vectors coming from q_1 and q_2.

We add a flag in the token embedding of TF to recognize the decision tokens of TF_1 and TF_2. The flag is in the coordinate f_c.

$$\mathsf{TE}(\sigma)_{f_c} = \begin{cases} 1 & \text{if } \sigma = q_{accept} \\ 0 & \text{otherwise} \end{cases}$$

Then, adding $h_{|\alpha|+T_1(|\alpha|)}$ to $h_{|\alpha|+T_1(|\alpha|)+T_2(\alpha)}$, we have 1 or 2 in the coordinate f_c iff TF_1 or TF_2 accepted α. Now we make all of $h_{|\alpha|+T_1(|\alpha|)+T_2(\alpha)}$ 0, except the f_c coordinate (using Section D.3).

Before going to the OUTPUT layer we move the value of f_c to the coordinate that will project to the probability of q_{accept} and add some number larger than 0 but smaller than 1 to the coordinate that will define the probability of q_{reject}. Therefore, the token with the maximum probability will be q_{accept} iff TF_1 or TF_2 accepted α, and it will be q_{reject} in the other case.

Then, TF computes the union of the languages of TF_1 and TF_2. The number of steps of CoT that it needs is exactly the ones made by TF_1 plus the ones made by TF_2 plus 1 extra for taking the disjunction, thus it has the same asymptotic cost as the maximum of TF_1 and TF_2. The embedding dimension of TF is the same as the dimension of TF_1 plus the dimension of TF_2 plus some constant number of flags, thus in $O(\cdot)$ notation it is the same as the maximum of TF_1 and TF_2.

Therefore, the language computed by the family of transformers $\{\mathsf{TF}_n\}_{n\in\mathbb{N}}$ is in $\mathsf{CoT}[T(n), d(n)]$. Thus, $\mathcal{L}_1 \cup \mathcal{L}_2 \in \mathsf{CoT}[T(n), d(n)]$.

F.3 Error reduction for BPCoT

Recall the definition of language recognition by probabilistic transformers. We say that a family of probabilistic transformers $\{\mathsf{TF}_n\}_{n\in\mathbb{N}}$ recognizes a language $\mathcal{L}$ when

$$\alpha \in \mathcal{L} \implies \Pr[\mathsf{TF}^*_{|\alpha|}(\alpha) = q_{accept}] > 2/3$$
$$\alpha \notin \mathcal{L} \implies \Pr[\mathsf{TF}^*_{|\alpha|}(\alpha) = q_{reject}] > 2/3$$

The choice of the constant $2/3$ may seem somewhat arbitrary. We now show that we can replace $2/3$ with any constant in the interval $(1/2, 1)$.

Let $\{\mathsf{TF}_n\}_{n\in\mathbb{N}}$ be a family of probabilistic transformers such that $\Pr[\mathsf{TF}^*_{|\alpha|}(\alpha) =$ the token consistent with $\mathcal{L}(\alpha)] > c$ for some fixed c in the interval $(1/2, 1)$ and show that it is equivalent to a family $\{\mathsf{TF}'_n\}_{n\in\mathbb{N}}$ that answers correctly with probability $d < 1$.

For each TF_n we construct TF'_n such that for all words of length n, $\Pr[\mathsf{TF}'^*_n(\alpha) =$ the token consistent with $\mathcal{L}(\alpha)] > d$. First, if $d \leq c$, then $\mathsf{TF}_n = \mathsf{TF}'_n$. Let us explore the case where $c < d$.

We construct TF'_n based on TF_n using a Majority Vote simulation.

The idea is similar to the disjunction of languages. TF'_n runs the transformer TF_n sequentially r times and then analyzes whether the majority of the results accept.

The construction presented in Appendix F.2 for the disjunction of two deterministic transformers can be easily extended to run a larger number of transformers sequentially. This construction also applies for probabilistic transformers.

To compute the disjunction, we analyzed the coordinate f_c at the last step of the computation. It worked as a counter of how many transformers accepted the word. We replicate this strategy to check whether the majority of the r runs of TF_n accepted the word, i.e., $f_c > r/2$. If that is the case, we force TF'_n to accept; otherwise, we force it to reject.

By Chernoff's bound, it holds that $r = O(1)$ (since both c and d are fixed). Thus, only a constant number of runs are needed for the construction to work properly.

For $|\alpha| = n$, let $T(n)$ be the number of steps the family $\{\mathsf{TF}_n\}_{n\in\mathbb{N}}$ needs to compute $\mathcal{L}(\alpha)$ and $d(n)$ its embedding dimension. Then, the family $\{\mathsf{TF}'_n\}_{n\in\mathbb{N}}$ takes $O(r \cdot T(n))$ steps to compute $\mathcal{L}(\alpha)$ and uses embedding size $O(r \cdot d(n))$. Thus, the $\mathsf{BPCoT}[T(n), d(n)]$ classes are invariant to the choice of the constant in the definition of language recognition by probabilistic transformers.

G SAMPLING THE DISTRIBUTION WITH A $\mathsf{BPAC}^0[1]$ CIRCUIT

Let p be the probability distribution computed by the distribution generator algorithm of a probabilistic, constant-depth, polynomial-embedding-dimension transformer. In this section we show that sampling p can be done with a constant-depth and polynomial-size random circuit using only a constant number of random gates.

We can sample from the distribution p by drawing a uniform random variable on the interval $[0, 1)$. We refer to the value of this variable as the seed. This is because we can partition $[0, 1)$ into disjoint intervals, one per token, each of length equal to the token's probability. Let p be the probability computed by the distribution generator. Let $\Sigma = \{\sigma_1, \ldots, \sigma_{|\Sigma|}\}$. Then the ranges are defined as $\text{range}(\sigma_i) = [\sum_{j=1}^{i-1} p(\sigma_j), \sum_{j=1}^{i} p(\sigma_j))$.

For example, if $\Sigma = \{t_1, t_2, t_3\}$ and p is such that $p(t_1) = 0.33$, $p(t_2) = 0.22$, $p(t_3) = 0.45$, then the ranges would be $[0, 0.33)$ for t_1, $[0.33, 0.55)$ for t_2, and $[0.55, 1)$ for t_3.

Observe that although the numbers $p(\sigma_j)$ are representable in $\mathbb{F}_{e,s}$, the summations are not performed with iterative rounding. They are computed with the maximum precision needed to avoid rounding error; this precision can be bounded as a function of $|\Sigma|$, e and s alone, independently of the values of $p(\sigma_i)$.

Since the precision for the numbers is fixed and the number of terms of the summations are bounded by the size of the alphabet, the ranges can be computed by circuits in AC^0.

Let us analyze the seed. It is just a random number in the range $[0, 1)$. We can interpret it as a binary string. How many bits of the seed are needed to determine which range it belongs to? It suffices to examine as many bits of the seed as the logarithm of the smallest probability.

Since p comes from the distribution generator, for all tokens, $p(\sigma)$ has to be representable in $\mathbb{F}_{e,s}$. Then, the smallest non-empty range has size at least ϵ, where ϵ is the smallest positive number representable in $\mathbb{F}_{e,s}$. Let r be the number of bits needed for representing ϵ in binary. Therefore, knowing the first r bits of the binary representation of the seed is enough to sample p. Thus, we only need r random bits for sampling p.

Note that r only depends on the sizes (e and s) of the significand and exponent of $\mathbb{F}_{e,s}$. Thus, r is constant for constant-precision transformers.

Observe that once the ranges are computed, they act as a lookup table with one entry for each number in $[0, 1)$ whose binary representation has r bits. Since r is a constant, the ranges can be interpreted as a constant-size lookup table.

Therefore, since the ranges can be computed by AC^0 circuits and the lookup table has constant size, sampling from p can be performed by a circuit in $\mathsf{BPAC}^0[1]$.

H SIMULATION OF RANDOMIZED CIRCUITS BY PROBABILISTIC TRANSFORMERS

Let $\{C_n\}_{n\in\mathbb{N}}$ be a family of randomized circuits of polynomial size. We construct a family of probabilistic transformers $\{\mathsf{TF}_n\}_{n\in\mathbb{N}}$ such that each TF_n simulates C_n. Recall that a randomized circuit C_n accepts α when $\Pr_{r_1\ldots r_{f(n)}}[C'_n(\alpha, r) = 1] > 2/3$, that is, the deterministic circuit C'_n accepts (α, r) for $2/3$ of the possible values of r. Our construction first generates r at random and then simulates the deterministic circuit using the construction of Li et al. (2024) for their Theorem 3.3.

Let TF'_n be the deterministic transformer simulating C'_n in $|C'_n|$ steps of CoT. Then, $\mathsf{TF}'_n(\alpha r) = C'_n(\alpha, r)$.

We assume TF'_n to be normalized. Then, its OUTPUT matrix is a projection.

TF'_n computes the i-th gate when the tape contains $i-1$ elements, i.e., when the values of all gates with lower index have been computed. It computes C'_n starting by the gate with index $n+f(n)+1$. By our assignment of the indices, the first Boolean gate is the one with index $n + f(n) + 1$. Thus, TF'_n begins meaningful computation once its tape contains $n + f(n)$ printed tokens.

We construct TF_n that simulates C_n by modifying TF'_n. Observe that since TF'_n is normalized, the change in token selector from a deterministic to a probabilistic selector does not affect its behavior. We define TF_n exactly as the probabilistic version of TF'_n, with one modification: it ignores the distributions generated by TF'_n in the first $f(n)$ steps of CoT.

The value in each position in the tape of TF_n corresponds to the value of a gate in C_n. The positions may hold the value 1, the value 0, or remain empty if their value has not yet been computed. The first n positions are the bits of α. The following $f(n)$ are the bits of r. TF'_n can generate the tokens for the positions after $f(n)$, i.e., compute the values of the Boolean gates. The probability distributions TF'_n generates in the first $f(n)$ steps, i.e., the distributions corresponding to the tokens representing r, do not correspond to what we want (since from TF'_n's perspective, they are part of the input). Thus, using the PE, we flag positions n through $n + f(n) - 1$ so that a constant function can be applied to them at the end of the body, forcing the probability distribution for tokens at positions $n + 1$ through $n + f(n)$ to be the uniform distribution over $\{0, 1\}$.

Then, after $f(n)$ steps of CoT, in the tape of TF_n there is written α followed by $f(n)$ random bits. From this point onward, TF'_n simulates $C'_n(\alpha, r)$.

TF_n accepts α if it accepts for at least $2/3$ of the possible runs. TF_n uses randomness only to generate the first $f(n)$ tokens. It generates them following a uniform distribution. Then, $\Pr[\mathsf{TF}_n(\alpha) = 1] = \Pr_r[\mathsf{TF}'_n(\alpha r) = 1] = \Pr_r[C'(\alpha, r) = 1] = \Pr[C_n(\alpha) = 1]$. Therefore, $\Pr[\mathsf{TF}_n(\alpha) = 1] > 2/3$ if and only if $\Pr[C_n(\alpha) = 1] > 2/3$.

I SENSITIVITY ARGUMENTS

Proof. Let $\mathcal{L} \in \mathsf{BPCoT}[T(n), \mathsf{poly}(n)]$, and fix a transformer family $\{\mathsf{TF}_n\}_{n\in\mathbb{N}}$ witnessing this membership. We write $q = |\Sigma|$, and assume without loss of generality that each transformer TF_n halts after exactly $T(n)$ steps for every input of size n (see Appendix D.7). Also, recall that by Lemma 4.1 each sampling step can be simulated with $O(1)$ random bits.

Given a sequence $x \in \Sigma^n$ and $y \in \Sigma^i$ with $i < T(n)$ there is a AC^0 circuit $G_{n,i}$ that computes the mapping

$$\mathsf{TF}_n(xy, r) = G_{n,i}(x, y, r),$$

where the bits r represent random bits used to simulate the sampling. Set $b = \max\{1, \lfloor \log_q n \rfloor\}$, and partition the $T(n)$ steps of CoT into consecutive blocks of length at most b. Consider a block of length $\ell \leq b$ starting after step i, and let $y \in \Sigma^i$ be the result of the CoT up this step For each candidate continuation $t = (t_1, \ldots, t_\ell) \in \Sigma^\ell$, define

$$H_{i,t}(x, y, R) = \bigwedge_{j=1}^{\ell} \left[G_{n,i+j}(x, y, t_1, \ldots, t_{j-1}, R_{i+j}) = t_j \right],$$

where R is a stream of $n + i + \ell$ random bits. Note that the ℓ conjuncts in this expression can be evaluated in parallel, each in a AC^0 circuit. Then, $H_{i,t}$ can also be computed in AC^0 by computing the conjunction of all these results. Let $S(n) = O(\mathsf{poly}(n))$ bound the size of this circuit[11].

For every fixed input x, prefix y, and assignment of the random bits R, exactly one candidate t satisfies $H_{i,t}(x, y, R) = 1$. Thus, for every $1 \leq j \leq \ell$ and $v \in \Sigma$, the predicate

$$[y_{i+j} = v] = \bigvee_{\substack{t\in\Sigma^\ell \\ t_j = v}} H_{i,t}(x, y, R)$$

can be used to compute the next ℓ tokens of the CoT with a circuit of constant depth and size $O(\ell q^\ell (S(n) + 1)) = O(\mathsf{poly}(n))$.

By composing these circuits we can generate the complete CoT for any input string x and fixed random seed R. The whole circuit has depth

$$O\left(\left\lceil \frac{T}{b} \right\rceil \right) = O\left(1 + \frac{T}{\log n} \right),$$

and its size is at most

$$O\big(Tq^b(S(n) + 1)\big) = O(\mathsf{poly}(n))$$

In particular, if $T = \log^k(n)$ the resulting depth is $O(\log^{k-1}(n))$.

Finally, to simulate the computation of TF_n we run many copies of this circuit with different random seeds. It is easy to show using standard tools that running $O(1)$ copies of the circuits with different seeds and taking a majority vote the result of the computation is the correct one with probability above $2/3$. Moreover, we can use an AC^0 circuit to compute majority in this case since there are only $O(1)$ values. □

[11] Note that the size can be bounded by an expression that only depends on n because $i \leq T(n)$ and $\ell \leq b \leq \log(n)$.

J SENSITIVITY PROOFS

For a Boolean function $f : \{0,1\}^n \to \{0,1\}$, its average sensitivity is defined as

$$\mathrm{AS}(f) = \sum_{i=1}^{n} \Pr[f(X) \neq f(X^{\oplus i})],$$

where X is sampled uniformly over $\{0,1\}^n$ and $x^{\oplus i}$ denotes x with its ith bit flipped. This same definition can be applied to function whose codomain is $\{-1,1\}$. In particular, $\mathrm{AS}(f) = \mathrm{AS}(1-2f)$.

Fix some family of transformers $\{\mathsf{TF}_n\}_{n\in\mathbb{N}}$ that performs $T(n) = O(\mathsf{poly}(n))$ steps of CoT (and assume without loss of generality that for all inputs it uses all the $T(n)$ steps), and that uses the symbol 1 to accept and 0 for reject. As before, we write $R = R_1 \dots R_{T(n)}$ for the sequence of blocks random bits required to simulate the sampling at the end of each inference step (where R_i is used for the ith sampling).

The sensitivity of a function is related to its Fourier characters. For $h : \{0,1\}^n \to \mathbb{R}$ and $S \subseteq \{1,\dots,n\}$, we write

$$\widehat{h}(S) = \mathbb{E}_X\left[h(X)(-1)^{\sum_{i\in S} X_i}\right], \qquad W_{\geq j}(h) = \sum_{|S|\geq j} \widehat{h}(S)^2.$$

By Parseval's identity, it holds that

$$\mathrm{AS}(h) = \sum_{S} |S|\widehat{h}(S)^2 = \sum_{j=1}^{n} W_{\geq j}(h). \tag{5}$$

We now prove Lemma 4.2. Fix n and r, and write $T = T(n)$ and $q = |\Sigma|$. Let $G_{n,i}(x, y, r')$ denote the next-token circuit. For each candidate string of CoT $t = (t_1, \dots, t_T) \in \Sigma^T$, define

$$h_{r,t}(x) = [t_T = q_{accept}] \wedge \bigwedge_{j=1}^{T} \left[G_{n,j-1}(x, t_1, \dots, t_{j-1}, r_j) = t_j\right].$$

Since $T \leq \mathsf{poly}(n)$, the circuit computing $h_{r,t}$ has polynomial size and constant depth (note that each conjunct can be evaluated in parallel in AC^0).

For every x and r there is only one t satisfying $h_{r,t}(x)$. Hence

$$\mathsf{TF}_n(x,r)(x) = \sum_{t\in\Sigma^T} h_{r,t}(x).$$

Set $F_r = 1 - 2\mathsf{TF}_n(x,r)$ and $B_{r,t} = 1 - 2h_{r,t}$. For every nonempty S, the preceding identity gives

$$\widehat{F}_r(S) = \sum_{t\in\Sigma^T} \widehat{B}_{r,t}(S).$$

The Fourier-tail bound for AC^0 circuits from Tal (2017) state that there is a function $b(n) \geq 1$, with $b(n) = O((\log n)^C)$ for some constant C, such that

$$W_{\geq j}(B_{r,t}) \leq 2\, 2^{-j/b(n)}.$$

For each seed r and string r the same bound can be applied. Thus, using Cauchy-Schwarz we can conclude, every $1 \leq j \leq n$, that

$$\begin{aligned} W_{\geq j}(F_r) &= \sum_{|S|\geq j} \left(\sum_{t\in\Sigma^T} \widehat{B}_{r,t}(S)\right)^2 \\ &\leq q^T \sum_{t\in\Sigma^T} W_{\geq j}(B_{r,t}) \leq 2q^{2T} 2^{-j/b(n)}. \end{aligned}$$

Note also that $W_{\geq j}(F_r) \leq 1$ by Parseval's identity. Put $b = b(n)$ and $K = \lceil b(1 + 2T \log_2 q) \rceil$. Using equation 5 and summing the geometric tail, we obtain

$$\begin{aligned}\mathrm{AS}(\mathsf{TF}_n(x, r)) = \mathrm{AS}(F_r) &= \sum_{j=1}^{n} W_{\geq j}(F_r) \\ &\leq K + \sum_{j=K+1}^{\infty} 2q^{2T} 2^{-j/b} \\ &\leq K + \frac{1}{2^{1/b} - 1} = O((T+1)b).\end{aligned}$$

Since q is fixed and $b = O((\log n)^C)$, this proves the lemma.

We now prove Theorem 4.5. Fix an input x and a coordinate i such that $f_n(x) \neq f_n(x^{\oplus i})$. If we run both transformers with some random string R, then each computation is incorrect with probability at most ε, so the union bound shows that both are correct with probability at least $1 - 2\varepsilon$. Whenever both are correct, their outputs differ. Consequently, for every x and i,

$$\Pr_R[\mathsf{TF}_n(x, r)(x) \neq \mathsf{TF}_n(x, r)(x^{\oplus i})] \geq (1 - 2\varepsilon)\mathbf{1}\{f_n(x) \neq f_n(x^{\oplus i})\}.$$

Averaging uniformly over all possible values of x summing over i we get

$$(1 - 2\varepsilon)\,\mathrm{AS}(f_n) \leq \mathbb{E}_R[\mathrm{AS}(\mathsf{TF}_n(x, r))] \leq c(T(n) + 1)(\log n)^C,$$

where the last inequality follows from Lemma 4.2.

For the last statement related to BPCoT, take $\varepsilon = 1/3$.

K EXTENDING THE SENSITIVITY ARGUMENTS

We first prove Proposition 4.1.Let $B_m(z, r) = \mathsf{TF}_{N(m)}(\pi_m(z), R)$. This computation generates at most $T(N(m))$ tokens and has error at most ε on every input z. If $G_{N,i}(x, y, r)$ is the next-token circuit after a prefix y of length i, its counterpart in this computation is

$$\widetilde{G}_{m,i}(z, y, r) = G_{N(m),i}(\pi_m(z), y, r).$$

After fixing y and r, these circuits have constant depth and polynomial size in m, with common bounds over all prefixes, positions, and random bits. This follows by composition and the polynomial bound on $N(m)$.

The proof of Lemma 4.2 uses only these circuit properties. It therefore applies to B_m, giving

$$\mathrm{AS}(B_m(\cdot, r)) \le c\big(T(N(m)) + 1\big)(\log m)^C$$

for every fixed random tape r. Applying the same random tape to two neighboring inputs and using the pointwise error guarantee, as in the proof of Theorem 4.5, yields

$$(1 - 2\varepsilon)\,\mathrm{AS}(g_m) \le \mathbb{E}_R[\mathrm{AS}(B_m(\cdot, R))].$$

Combining the inequalities proves the result.

Now, we show Corollary 4.4. Let $m = \lfloor (n-2)/2 \rfloor$. Given $z \in \{0,1\}^m$, construct a graph G_z with vertices

$$(i, b), \qquad 0 \le i \le m, \quad b \in \{0, 1\},$$

and edges

$$(i-1, b) \longrightarrow (i, b \oplus z_i), \qquad 1 \le i \le m, \quad b \in \{0, 1\}.$$

Add isolated vertices if necessary to obtain exactly n vertices, and label $(0, 0)$ and $(m, 1)$ as s and t. The path starting at s reaches $(i, z_1 \oplus \cdots \oplus z_i)$ in layer i. Hence

$$\mathsf{Reach}(G_z) = \mathrm{PARITY}_m(z).$$

Every adjacency entry of G_z is a constant, z_i, or $\neg z_i$, so the map $z \mapsto G_z$ is an AC^0 reduction. Since $\mathrm{AS}(\mathrm{PARITY}_m) = m$ and $m = \Theta(n)$, Proposition 4.1 gives

$$\frac{m}{3} \le c(T(N) + 1)(\log m)^C,$$

which proves the bound.

L SIMULATION OF DETERMINISTIC TRANSFORMERS BY PROBABILISTIC TRANSFORMERS

In this section, we give a more detailed construction for the proof of Theorem 5.1.

Let TF be a deterministic transformer. We construct a probabilistic transformer TF' that accepts the same words in the same number of steps of CoT.

As per Appendix E, for every deterministic transformer, we can construct another one equivalent such that probability distribution generated by it in each step satisfies the property that there exists $\sigma \in \Sigma$ such that:

$$p(\tau) = \begin{cases} 1 & \text{if } \tau = \sigma \\ 0 & \text{otherwise} \end{cases}$$

Said construction enquirers only a constant blow up in embedding size. Lets us call TF_n to the construction satisfying the property equivalent to TF.

Then, we define TF' exactly as TF_n, except for the token selector which by definition is the probabilistic.

TF_n and TF' behave exactly the same, since sampling this distribution p always returns the token with probability 1, the same token that $\arg\max$ would return. Next, the deterministic token selector and the probabilistic token selector behave the same. Thus, for all i and α, $\mathsf{TF}^i(\alpha) = \mathsf{TF}'^i(\alpha)$. In particular, this means that $\mathsf{TF}^*(\alpha) = \mathsf{TF}'^*(\alpha)$. Therefore, TF and TF' accept the same words (and in the same number of steps). By construction, the embedding size of TF and TF' is the same.