Can Transformers Reason Logically? A Study in SAT Solving

Leyan Pan [1] **Vijay Ganesh** [1 *] **Jacob Abernethy** [1 2 *] **Chris Esposo** [1] **Wenke Lee** [1]

Abstract

We formally study the logical reasoning capabilities of decoder-only Transformers in the context of the boolean satisfiability (SAT) problem. First, we prove by construction that decoder-only Transformers can decide 3-SAT, in a non-uniform model of computation, using backtracking and deduction via Chain-of-Thought (CoT). Second, we implement our construction as a PyTorch model with a tool (`PARAT`) that we designed to empirically demonstrate its correctness and investigate its properties. Third, rather than *programming* a transformer to reason, we evaluate empirically whether it can be *trained* to do so by learning directly from algorithmic traces ("reasoning paths") from our theoretical construction. The trained models demonstrate strong out-of-distribution generalization on problem sizes seen during training but have limited length generalization, which is consistent with the implications of our theoretical result.

1. Introduction

Transformer-based large language models (LLMs, Vaswani et al. (2017)) have demonstrated strong performance on tasks that seem to demand complex reasoning, especially with Chain-of-Thought (CoT, Wei et al. (2022); OpenAI (2024); DeepSeek-AI et al. (2025)). However, they often face challenges in reliable multi-step logical reasoning, hallucinating logically flawed or factually incorrect conclusions. Consequently, many researchers reject the idea that LLMs can reason (Kambhampati et al., 2024), and researchers continue to disagree on the precise definition of "reasoning" in the context of LLMs. Furthermore, there is little understanding of the fundamental limitations of the reasoning capabilities of Transformer models.

[*]Equal contribution [1]College of Computing, Georgia Institute of Technology, Atlanta, GA, USA [2]Google Research, Atlanta, USA. Correspondence to: Leyan Pan <leyanpan@gatech.edu>.

Proceedings of the 42^{nd} International Conference on Machine Learning, Vancouver, Canada. PMLR 267, 2025.

This paper focuses on the *deductive logical* reasoning capability of the Transformer model in a restricted but simple and mathematically precise setting, namely, the Boolean satisfiability problem (SAT, Cook (1971)). We view deductive reasoning as the process of systematically drawing valid inferences from existing premises and assumptions. Boolean SAT solving captures the essence of deductive logical reasoning because: 1) Boolean logic lies as the foundation of all logical reasoning, and 2) many modern SAT solvers are inherently formal deductive systems that implement the resolution proof system. Its NP-Completeness also necessitates multiple rounds of trial and error, which is critical for solving complex problems.

We prove by construction that decoder-only Transformers can decide 3-SAT instances with CoT (in a non-uniform computational setting):

Theorem 1.1 (Informal version of Theorem 4.5)**.** *For any $p, c \in \mathbb{N}^+$, there exists a decoder-only Transformer with $O(p^2)$ parameters that can decide all 3-SAT instances of at most p variables and c clauses using Chain-of-Thought.*

The "non-uniform" nature of this result means that the Transformer's parameters are dependent on the problem size (p), in contrast to a single, universal model that handles arbitrary input lengths. An insight from our construction is that this practical difficulty in achieving length generalization stems from numerically approximating precise attention patterns with softmax, where errors can compound on larger instances and affect performance (Figure 5). This provides a theoretical explanation for the length generalization limitations observed when training LLMs on reasoning tasks.

We illustrate the CoT our construction uses to solve 3-SAT instances in Figure 1. The Transformer model simulates logical assumption, deduction, and backtracking by generating new tokens and ultimately outputs a `SAT`/`UNSAT` label as the result of the 3-SAT decision problem. Our theoretical construction shows that, for any chosen instance size, a standard softmax-attention Transformer with size-dependent parameters can decide 3-SAT, requiring only a single forward pass to perform logical deduction over *all* clauses under the current variable assignments (see Lemma 4.8).

To empirically verify and investigate our construction, we design a tool (`PARAT`) that instantiates the weights of Trans-

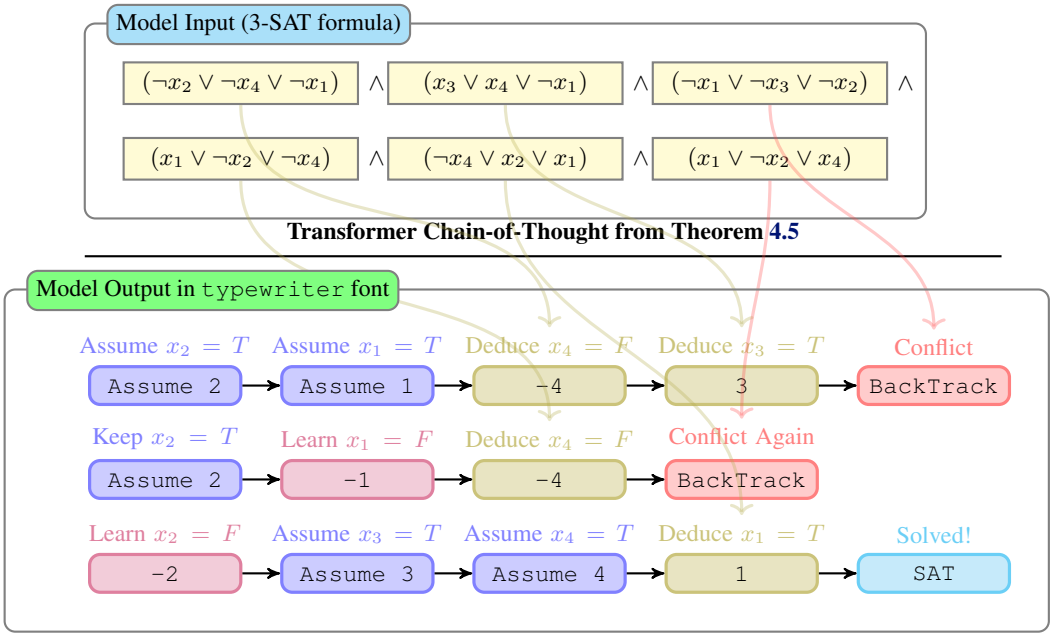

Figure 1. Visualization of the Chain-of-Thought (CoT) process used by our model to solve an example 3-SAT formula described in Theorem 4.5. The model autonomously performs trial-and-error reasoning, making multiple attempts and backtracking upon encountering conflicts. Here, T represents *True* and F represents *False*. Tokens in `typewriter` font denote the CoT generated by the model.

former models based on NumPy code specifying the desired behavior. With `PARAT`, we implemented the construction as a PyTorch Transformer model and empirically validated its correctness on random 3-SAT instances.

Additionally, we perform training experiments to demonstrate that Transformers can effectively learn from the deductive reasoning and backtracking process encoded as CoT. We show that trained Transformer models can generalize between SAT instances generated from different distributions within the same number of variables p. However, LLMs trained on SAT instances with CoT still struggle to solve instances with an unseen number of variables, demonstrating limitations in learning length-generalizable reasoning. These experimental results support our theoretical predictions.

Contributions First, we prove by construction that in a non-uniform model of computation, decoder-only Transformers can decide 3-SAT using backtracking and deduction via Chain-of-Thought (CoT). We show that Transformers can perform logical deduction on all conditions (clauses) in parallel instead of checking each condition sequentially. Nevertheless, the construction requires exponentially many CoT steps in the worst case, as implied by the NP-hardness of SAT, although it requires much fewer steps in most instances.

We design `PARAT`, a tool to instantiate Transformer model weights that implement specifications written in NumPy-style code. We empirically demonstrate that the instantiated Transformer corresponding to our theoretical construction can perfectly solve 3-SAT instances.

Finally, our supporting training experiments suggest that training on our theoretical CoT of 3-SAT reasoning traces allows Transformer models to achieve out-of-distribution generalization within the same input lengths, but fail to generalize to larger instances.

2. Related Work

Transformers and P and P/poly Problems. This line of research focuses on what types of computation can Transformer models simulated by providing theoretical constructions of Transformer models that can solve well-defined computational problems. The seminal work of Liu et al. (2023a) showed that Transformers can simulate semiautomata using a single pass over only a logarithmic number of layers w.r.t. the number of states. Yao et al. (2021) demonstrated that transformers can perform parentheses matching of at most k types of parentheses and D appearance of each ($\mathrm{Dyck}_{k,D}$) with $D+1$ layers.

However, the computation power of one pass of the Transformer model is fundamentally limited (Merrill et al., 2021; Merrill & Sabharwal, 2023), and the success of Chain-of-Thought (CoT) reasoning has sparked research on how CoT can improve upon the expressiveness of Transformer models. Pérez et al. (2019) proved that Transformers can emulate

the execution of single-tape Turing machines. Giannou et al. (2023) showed that Transformers can recurrently simulate arbitrary programs written in a one-instruction-set language. Li et al. (2024) proved that Transformers can simulate arbitrary boolean circuits using CoT by representing the circuit in the positional encoding. In particular, transformers can decide all problems in $\mathrm{P}/\mathrm{poly} \supseteq \mathrm{P}$ with polynomial steps of CoT. Merrill & Sabharwal (2024) showed that Transformers with saturated attention can decide all regular languages with a linear number of CoT tokens and decide all problems in P with a polynomial number of CoT tokens. (Feng et al., 2023) shows that Transformer CoT can perform integer arithmetic, solve linear equations, and perform dynamic programming for the longest increasing subsequence and edit distance problems.

How our work differs. We focus on 3-SAT, which is an NP-complete problem. It is widely believed that P is a strict subset of NP, and it is not known whether NP is a subset of P/poly. In other words, our results are not comparable to these earlier results.

Turing Completeness of Transformers. Meanwhile, Pérez et al. (2019), Li et al. (2024), and Merrill & Sabharwal (2024) also showed that Transformers can simulate single-tape Turing Machines (TM) with CoT and can theoretically be extended to arbitrary decidable languages. However, these constructions require at least one CoT token for every step of TM execution.

How our work differs. Our theoretical construction demonstrates that, for certain classes of formal reasoning problems, Transformers can simulate algorithmic reasoning traces at an abstract level with *drastically reduced number of CoT tokens* compared to step-wise emulation of a single-tape TM. At each CoT step, our construction performs deductive reasoning over the formula in parallel while any single-tape TM must process each input token sequentially. Furthermore, the CoT produced by our theoretical construction abstractly represents the human reasoning process of trial and error, as demonstrated in Figure 1.

Formal Logical Reasoning with LLMs Several studies also evaluate pretrained LLMs' *formal* and *algorithmic* reasoning abilities, finding that they perform well on a few reasoning steps but struggle as the required steps increase. ProofWriter (Tafjord et al., 2021), ProntoQA (Saparov & He, 2023; Saparov et al., 2023), FOLIO (Han et al., 2024), SimpleLogic (Zhang et al., 2023b), and RuleTaker (Clark et al., 2021) encodes formal logical reasoning as natural language problems to test general purpose LLMs on multi-step reasoning. NPHardEval (Fan et al., 2024) compiles a benchmark of P and NP-Hard problems and tests a variety of pre-trained LLMs. Liu et al. (2023b) evaluates code execution capabilities, Chen et al. (2024) measures capabilities to solve propositional and first-order logic satisfiability, and Hazra et al. (2024) investigates pretrained LLM's capability of solving SAT instances from the perspective of phase transitions.

A related line of work uses formal symbolic logic to enhance the capabilities of LLMs with CoT. LogicLM (Pan et al., 2023) and SymbCoT (Xu et al., 2024) integrate symbolic expressions of first-order logic with CoT prompting and invoke solvers to provide feedback the LLM reasoner. Ryu et al. (2025) uses divide and conquer to improve upon the above works in terms of translation accuracy. Jha et al. (2024) uses symbolic logic solvers to provide reinforcement learning rewards to improve LLM reasoning. Beyond LLMs, NeuroSAT (Selsam et al., 2019), MatSAT (Sato & Kojima, 2021), and SATformer (Shi et al., 2022) train different neural networks to learn SAT-solving.

How our work differs. Our work focuses on the theoretical capabilities of Transformer models rather than practical pretrained LLMs and can be viewed as building a theoretical foundation for these results.

Compilation of Transformer Weights. Further, prior work on the theoretical construction of Transformer models rarely provides practical implementations. Notably, Giannou et al. (2023) provides an implementation of their Transformer construction and demonstrates its execution on several programs. However, the model is initialized "manually" using prolonged sequences of array assignments. Lindner et al. (2023) released Tracr, which compiles RASP (Weiss et al., 2021) programs into decoder-only Transformer models. RASP is a human-readable representation of a subset of operations that Transformers can perform via self-attention and MLP layers. While having related functionalities, our tool has different goals than Tracr and bears multiple practical advantages for implementing complex constructions, which we detail in Appendix D.2.

3. Preliminaries

This section reviews the Boolean Satisfiability (SAT) problem, 3-SAT, and the classical DPLL search procedure. We provide a brief overview of the important notations used in this work. For a more detailed introduction to SAT-solving, we recommend to interested readers the first chapters of (Biere et al., 2009). We assume familiarity with the high-level components of Transformer models. For details on the mathematical model for Transformers used in our theoretical results, please refer to Appendix C.1.

3.1. Boolean Satisfiability (SAT) in Conjunctive Normal Form (CNF)

Let $\mathrm{Var} = \{x_1, \ldots, x_p\}$ be a set of *Boolean variables*. A *literal* is either a variable x_v or its negation $\neg x_v$. The set of

all $2p$ literals is denoted

$$L = \{x_1, \neg x_1, \ldots, x_p, \neg x_p\}.$$

A *clause* $C \subseteq L$ is a finite disjunction ("OR") of literals, written $C = (\ell_1 \vee \cdots \vee \ell_k)$. A *formula in conjunctive normal form (CNF)* is a conjunction ("AND") of clauses

$$F = C_1 \wedge C_2 \wedge \cdots \wedge C_c.$$

An *assignment* $A \subseteq L$ is any set of literals that does not contain both x_v and $\neg x_v$. Intuitively $x_v \in A$ means $x_v =$ `True` and $\neg x_v \in A$ means $x_v =$ `False`. A clause C is *satisfied* by A (denoted $A \models C$) if $C \cap A \neq \varnothing$; a CNF formula F is *satisfied* by assignment A if every clause is satisfied by A. The SAT problem asks whether a satisfying assignment exists for a given F.

3-SAT. In *3-SAT* each clause contains at most three literals. 3-SAT is NP-complete; it is the canonical hard case for SAT and is the focus of our work.

3.2. DIMACS Encoding

To represent 3-SAT formulas as a sequence of tokens to Transformer models, we adopt the standard DIMACS format. Each literal is encoded as an integer: v for x_v and $-v$ for $\neg x_v$. Clauses are sequences of integers terminated by `0`. For example

$$(x_1 \vee \neg x_2) \wedge (\neg x_1 \vee x_2 \vee \neg x_3)$$

is written

```
1 -2 0 -1 2 -3 0
```

with an initial `[BOS]` and a closing `[SEP]` token in our model implementation.

3.3. Partial Assignments and Formula Reduction

SAT solving reason with *partial* assignments that leave some variables unassigned. Given a CNF F and a partial assignment A, the *reduction* of F by A—denoted $F|_A$—is obtained in two steps:

Delete satisfied clauses: remove any C_i with $C_i \cap A \neq \varnothing$.

Delete falsified literals: from each remaining clause delete any literal that is false under A (e.g. delete $\neg x_v$ if $x_v \in A$).

If a clause becomes empty, the entire formula is unsatisfiable under A. A *unit clause* is a clause with exactly one literal after reduction; its lone literal must be true in every extension of A.

Example. Let

$$F = (x_1 \vee x_2) \wedge (\neg x_1 \vee x_3) \wedge (x_2 \vee \neg x_3), \quad A = \{x_1\}.$$

After Step 1 the first clause is deleted (already true). Step 2 removes $\neg x_1$ from the second clause, giving the unit clause (x_3). Hence $F|_A = (x_3) \wedge (x_2 \vee \neg x_3)$ and x_3 is *forced* true by unit propagation.

3.4. The DPLL Search Procedure

The Davis–Putnam–Logemann–Loveland (DPLL) algorithm is a complete, backtracking SAT-solving procedure. Starting from a CNF formula F and the empty assignment $\emptyset$, at any moment it maintains a partial assignment A and the reduced formula $F|_A$. The main operations are

Decision: Choose an unassigned variable x_v and tentatively set it to either `True` or `False`; record the choice so it can be undone later.

Unit Propagation: Repeatedly adds forced literals from unit clauses of $F|_A$ to A.

Conflict/Backtrack: If an empty clause appears, undo decisions chronologically until a new value can be tried; if none remain, the instance is unsatisfiable.

The procedure terminates when either all clauses are satisfied (`SAT`) or a conflict arises at the root level (`UNSAT`).

3.5. Notation Used in Later Sections

We summarise the symbols that re-appear in the theoretical part of the paper:

$[p]$	$\{1, \ldots, p\}$, index set of variables.
C_i	i-th clause in a CNF formula.
$F = \bigwedge_{i=1}^{c} C_i$	CNF formula with c clauses.
$A \subseteq L$	partial assignment (no complementary literals).
$F\|_A$	reduction of F by A (definition above).
$A \models F$	A satisfies F.
$F \models \neg A$	F contradicts A (empty clause in $F\|_A$).
$F \wedge A \models_1 \ell$	ℓ is forced by unit propagation.

These notations are used verbatim in Section 4.

4. Transformers and SAT: Logical Deduction and Backtracking

This section presents and explains our main results on Transformers' capability in deductive reasoning and backtracking with CoT. To rigorously state our results, we first formally define decision problems, decision procedures, and what it means for a model to "solve" a decision problem using CoT:

Definition 4.1 (Decision Problem)**.** Let $\mathcal{V}$ be a vocabulary, $\Sigma \subseteq \mathcal{V}$ be an alphabet, $L \subseteq \Sigma^*$ be a set of valid input strings. We say that a mapping $f : L \to \{0, 1\}$ is a *decision problem* defined on L.

Definition 4.2 (Decision Procedure)**.** We say that an algorithm $\mathcal{A}$ is a decision procedure for the decision problem f, if given any input string x from L, $\mathcal{A}$ halts and outputs 1 if $f(x) = 1$, and halts and outputs 0 if $f(x) = 0$.

Definition 4.3 (Autoregressive Decision Procedure)**.** For

any map $M : \mathcal{V}^* \to \mathcal{V}$, which we refer to as an *autoregressive next-token prediction model*, and $\mathcal{E} = \{\mathcal{E}_0, \mathcal{E}_1\} \subset \mathcal{V}$, define procedure $\mathcal{A}_{M,\mathcal{E}}$ as follows: For any input $s_{1:n}$, run Algorithm 1 with M and stop tokens $\mathcal{E}$. $\mathcal{A}_{M,\mathcal{E}}$ outputs 0 if $s_{1:t}$ ends with $\mathcal{E}_0$ and $\mathcal{A}_{M,\mathcal{E}}$ output 1 otherwise. We say M *autoregressively decides* decision problem f if there is some $\mathcal{E} \subset \mathcal{V}$ for which $\mathcal{A}_{M,\mathcal{E}}$ decides f.

Definition 4.4 (3-SAT$_{p,c}$)**.** Let DIMACS(p, c) denote the set of valid DIMACS encodings of 3-SAT instances with at most p variables and c clauses with a prepended `[BOS]` token and an appended `[SEP]` token. Define 3-SAT$_{p,c}$: DIMACS$(p, c) \to \{0, 1\}$ as the problem of deciding whether the 3-SAT formula, encoded via DIMACS(p, c), is satisfiable.

With the above definition, we present the formal statement of our main result:

Theorem 4.5 (Decoder-only Transformers can solve SAT)**.** *For any $p, c \in \mathbb{N}^+$, there exists a Transformer model $M : \mathcal{V}^* \to \mathcal{V}$ that autoregressively decides 3-SAT$_{p,c}$ in no more than $p \cdot 2^{p+1}$ CoT iterations. M requires $L = 7$ layers, $H = 5$ heads, $d_{emb} = O(p)$, and $O(p^2)$ parameters.*

Remarks on Theorem 4.5

- Despite the high upper bound on CoT length, it's rarely reached in practice. In Figure 4 we show that the CoT length is no greater than $8p \cdot 2^{0.08p}$ for most formulas
- The worst-case CoT length is independent of the number of clauses c, which is due to the parallel deduction over all clauses within the Transformer construction.
- Positional encodings are not included in the number of parameters. The positional encoding at position i is the numerical value i at a particular dimension.
- Each param. can be represented with $O(p + \log c)$ bits

Proof Sketch We show our full construction and proof via simulation of the abstract DPLL (Nieuwenhuis et al., 2005) in Appendix C. The construction uses adapted versions of lemmas from Feng et al. (2023) as basic building blocks. Here we provide a proof sketch of the core steps and operations in our theoretical construction.

Step 1: Summarize clauses and assignments as binary vectors The Transformer first converts every clause C_i and the evolving partial assignment A into the binary encodings of Definition 4.6 (see Fig.2). These vectors are obtained inside the model by summing the one-hot literal embeddings between two separator tokens; no wording inside the definition is changed.

Definition 4.6 (Encoding of clauses and partial assignments, extending Sato & Kojima (2021))**.** The mappings $E, E_{\text{not-false}}, E_{\text{assigned}} : \mathcal{B} \to \mathbb{R}^{2p}$ encodes $B \in \mathcal{B}$ as

$$E(B)_v := \mathbf{1}_{x_v \in B} \qquad E(B)_{v+p} := \mathbf{1}_{(\neg x_v) \in B}.$$
$$E_{\text{not-false}}(B)_v := \mathbf{1}_{(\neg x_v) \notin B} \qquad E_{\text{not-false}}(B)_{v+p} := \mathbf{1}_{x_v \notin B}.$$
$$E_{\text{assigned}}(B)_v := E_{\text{assigned}}(B)_{v+p} := \mathbf{1}_{x_v \in B \vee (\neg x_v) \in B}.$$

Of the above encodings, $E(B)$ is calculated by using self-attention layers to sum up one-hot token representations within a clause/partial assignment, while both $E_{\text{not-false}}(B)$ and $E_{\text{assigned}}(B)$ can be computed via an affine transformation on $E(B)$.

Step 2: Parallel logical operations over all clauses.

We now show that the core logical operations of DPLL SAT solving can be computed using vector operations on the encoding of the clauses $\{C_1, \ldots, C_c\}$ and a partial assignment A over all clauses in parallel, a capability of Transformers that allows efficient logical reasoning over long contexts:

Lemma 4.7. *Let $F = \bigwedge_{i \in [c]} C_i$ be a 3-SAT formula over p variables $\{x_1, \ldots, x_p\}$ and c clauses $\{C_1, \ldots, C_c\}$. Let $A \subset L$ be a partial assignment defined on variables $\{x_1, \ldots, x_p\}$, then the following properties hold:*

1. *Satisfiability Checking:*
$$A \models F \iff \min_{i \in [c]} E(C_i) \cdot E(A) \geq 1.$$
2. *Conflict Detection:*
$$F \models \neg A \iff \min_{i \in [c]} E(C_i) \cdot E_{\text{not-false}}(A) = 0.$$
3. *Deduction: Let $D := \{l \in L \mid F \wedge A \models_1 l\}$ be the literals deducible from F given A via unit propagation. Then we can write $E(D)$ as*
$$\max\Big[\min\Big(\sum_{i \in [c]} E(C_i)\mathbf{1}_{\{E(C_i) \cdot E_{\text{not-false}}(A) = 1\}}, 1\Big) - E_{\text{assigned}}(A),\ 0\Big].$$
where max *and* min *are applied element-wise.*

Each of the above operations can be approximated by an attention head when given the clause and partial assignment encodings. We capture this idea in the following lemma:

Lemma 4.8 (Parallel Processing of Clauses, Informal)**.** *Let F be a 3-SAT formula over variables $\{x_1, \ldots, x_p\}$ with c clauses $\{C_1, \ldots, C_c\}$ and A a partial assignment defined on variables $\{x_1, \ldots, x_p\}$. Let*

$$X_{encoding} = \begin{bmatrix} 0 & 1 & 1 \\ E(C_1) & 0 & 1 \\ \vdots & \vdots & \vdots \\ E(C_c) & 0 & 1 \\ E(A) & 0 & 1 \end{bmatrix} \in \mathbb{R}^{(c+2)\times(2p+2)}$$

Then for any $1 > \epsilon > 0$, given X as input, there exists:

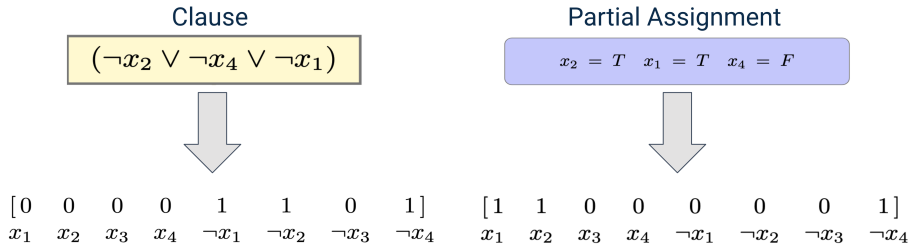

Figure 2. Illustration of $E(C)$ and $E(A)$ with $p = 4$.

- *An attention head that outputs $\mathbf{1}_{A \models F}$ with approximation error bounded by ϵ*
- *An attention head that outputs $\mathbf{1}_{F \models \neg A}$ with approximation error bounded by ϵ*
- *An attention head followed by an MLP layer that outputs $E(D)$ as defined Lemma 4.7 with $\|\cdot\|_\infty$ error bounded by ϵ, unless $F \models \neg A$*

All weight values are independent of F and A and are bounded by $O(poly(p, c, \log(1/\epsilon)))$

Step 3: Predicting the Next Token Each of the above results is stored in the hidden state vectors (i.e., residual stream (Elhage et al., 2021)) of the Transformer model. In the final layer, the prediction of the next token is determined by a series of conditionals based on a priority: If $A \models F$, then the next token is `SAT`. Otherwise, if $F \models \neg A$, output `UNSAT` if A contains decision literals and output `[BackTrack]` otherwise, etc. Collectively, the token predictions ensure that, when running Algorithm 1, the resulting transition from $s_{1:t}$ to $s_{1:t+1}$ at every timestep t emulates the abstract DPLL state transition system (Nieuwenhuis et al., 2005) and thus ensures that the procedure correctly decides 3-SAT. (See Appendix C.8.1 for details)

5. Implementing the Construction with `PARAT`

In the previous section, we presented a theoretical construction of a Transformer capable of solving SAT instances. However, it can be difficult to gain insights and fully verify its correctness without experimental interactions with the construction. To help address this, we introduce `PARAT` (short for ParametricTransformer), which instantiates Transformer weights based on high-level specifications written as NumPy code performing array operations.

Both `PARAT` and the specification it accepts are based on Python, and the syntax of the `PARAT` is a restricted subset of Python with the NumPy library. Every variable `v` in `PARAT` is a 2-D NumPy array of shape `n` × `d_v`, where `n` denotes the input number of tokens and `d_v` is the dimension of the `PARAT` variable `v`, which can be different for every variable.

A specification "program" in `PARAT` is composed of a linear sequence of statements (i.e., no control flow such as loops or branching based on `PARAT` variable values is allowed), where each statement assigns the value of an expression to a variable. Let `v_1`, `v_2`, ... denote `PARAT` variable names. Each statement involving `PARAT` variables must be one of the following: **(1) Binary operations** such as `v_1 + v_2`, `v_1 * v_2`, `v_1 - v_2`; **(2) Index operations** such as `v_1[v_2, :]` or `v_1[:, start:end]`, where `start, end` $\in [d_{\texttt{v\_1}}]$; or **(3) Function calls** from a predefined library of functions that take `PARAT` variables as input.

`PARAT` takes in a specification program as well as variable `out` of dimension V (size of vocabulary) and outputs a PyTorch `Module` object that implements a Transformer model as defined in Section 2. The following condition is satisfied: For any possible input sequence of tokens s in the vocabulary of length n, the token predicted by the Transformer model is the same as the token corresponding to `out[-1, :].argmax()` (i.e., the token prediction at the last position) when interpreting the specification using the Python interpreter with the NumPy library. We provide more details on our tool and the supported operations in section Appendix D.

5.1. Analysis of the Transformer Construction

With our tool, we successfully implemented our theoretical construction in Theorem 4.5 using the code in Appendix D.4 as a PyTorch model. We will refer to this model as the "compiled" model for the rest of the section. With a concrete implementation of our theoretical construction in PyTorch, we empirically investigate 3 questions (1) Does the compiled model correctly decide SAT instances? (2) How many steps does the model take to solve actual 3-SAT instances? (3) How does error induced by soft attention affect reasoning accuracy?

Evaluation Datasets We evaluate our models on randomly sampled DIMACS encoding of 3-SAT formulas. We focus on SAT formulas with exactly 3 literals in each clause, with the number of clauses c between $4.1p$ and $4.4p$, where p is

the number of variables.

It is well-known that the satisfiability of such random 3-SAT formulas highly depends on the clause/variable ratio, where a formula is very likely satisfiable if $c/p \ll 4.26$ and unsatisfiable if $c/p \gg 4.26$ (Crawford & Auton, 1996). This potentially allows a model to obtain high accuracy just by observing the statistical properties such as the c/p ratio. To address this, we constrain this ratio for all formulas to be near the critical ratio 4.26. Furthermore, our "marginal" datasets contain pairs of SAT vs UNSAT formulas that differ from each other by only a single literal. This means that the SAT and UNSAT formulas in the dataset have almost no statistical difference in terms of c/p ratio, variable distribution, etc., ruling out the possibility of obtaining SAT vs UNSAT information solely via statistical properties. We also use 3 different sampling methods to generate formulas of different solving difficulties to evaluate our model:

Marginal: Composed of pairs of formulas that differ by only one token.

Random: Formulas are not paired by differing tokens and each clause is randomly generated.

Skewed: Formulas where polarity and variable sampling are not uniform; for each literal, one polarity is preferred over the other. Some literals are also preferred over others.

We generate the above 3 datasets for each variable number $4 \leq p \leq 20$, resulting in 51 total datasets of 2000 samples each. Each sample with p variables contains $16.4p$ to $17.6p$ input tokens, which is at least 320 for $p = 20$.

Model Unless otherwise stated, the model we experiment with is compiled from the code in D.4 using `PARAT` with max number of variables $p = 20$, max number of clauses $c = 88$, and exactness parameter $\beta = 20$. The model uses greedy decoding during generation.

Accuracy Our compiled model achieves perfect accuracy on all evaluation datasets described above. This provides empirical justification for our theoretical construction for Theorem 4.5 as well as `PARAT`. This result is included in Figure 3 to compare with trained models.

How many steps? For all formulas we evaluated, the maximum CoT length is bounded by $8p \cdot 2^{0.08p}$, which is significantly less than the theoretical bound of $p \cdot 2^{(p+1)}$. This indicates that the model can use deduction to reduce the search space significantly. See appendix Figure 4.

Effect of Softmax Attention In our previous evaluations, we used a sufficiently large scaling factor β to ensure that the approximation error from the softmax is bounded and does not affect the final token output. In Figure 5 we show that, with an insufficient scaling factor, the compiled model would achieve perfect accuracy on smaller instances but degrade significantly on larger ones. This matches the commonly observed limitation in length generalization when training on smaller problem instances.

6. Can Transformer Learn SAT Solving?

Our previous sections showed that Transformer and weights exist for solving SAT instances using CoT with backtracking and deduction. However, it is unclear to what extent Transformers can learn such formal reasoning procedures by training on SAT formulas. Previously, Zhang et al. (2023a) showed that when using a single pass of a Transformer model (without CoT), Transformers fail to generalize to logical puzzles sampled from different distributions even when they have the same number of propositions.

This section provides proof-of-concept evidence that training on the CoT procedure with deduction and backtracking described in Figure 1 can facilitate out-of-distribution generalization within the same number of variables.

Datasets In Section 5.1 we introduced 3 different distributions over random 3-SAT formulas of varying difficulties. For training data, we use the same sampling methods, but instead of having a separate dataset for each variable number p, we pick 2 ranges $p \in [6, 10]$ and $p \in [11, 15]$, where for each sample a random p value is picked uniformly random from the range. Each formula with p variables contains $16.4p$ to $17.6p$ tokens. This results in 2×3 training datasets, each containing 5×10^5 training samples[1], with balanced SAT vs UNSAT samples. For each formula, we generate the corresponding CoT in the same format as Figure 1 using a custom SAT Solver. The evaluation data is exactly the same as Section 5.1.

Model and Training We use the LLaMa (Touvron et al., 2023) architecture with 70M and 160M parameters for the training experiments. We compute cross-entropy loss on every token in the CoT but not the DIMACS encoding in the prompt tokens. We provide further training details in Appendix A. We also permute the variable IDs for training samples to ensure that the model sees all possible input tokens for up to 20 variables.

Evaluation Criteria We evaluate our model using two criteria: SAT/UNSAT accuracy and full trace correctness. SAT/UNSAT accuracy evaluates the model's binary prediction based on the first token in $\{\texttt{SAT}, \texttt{UNSAT}\}$ generated by the model, compared against the ground truth satisfiability of the formula. If the model fails to generate $\{\texttt{SAT}, \texttt{UNSAT}\}$ within the context length, the prediction is considered incorrect, which can cause accuracy to drop significantly below 50%. Full trace correctness checks if *every* token gener-

[1]The number of training samples is negligible compared to the total number of possible formulas. There are more than p^{12p} 3-SAT formulas with p variables, which is $> 10^{56}$ for $p = 6$

		$p \in [6, 10]$			$p \in [11, 15]$		
		Marginal	Random	Skewed	Marginal	Random	Skewed
SAT vs UNSAT	**Marginal**	99.88%	99.99%	99.99%	99.82%	99.89%	99.81%
	Random	99.96%	100.00%	100.00%	99.11%	99.75%	99.55%
	Skewed	99.96%	100.00%	99.99%	99.41%	99.74%	99.48%
Full Trace Correct	**Marginal**	98.50%	97.33%	88.72%	98.66%	97.57%	86.06%
	Random	99.40%	99.04%	93.12%	98.56%	97.99%	91.70%
	Skewed	99.38%	99.16%	97.72%	97.02%	95.98%	91.51%

Table 1. Average accuracies (%) of SAT/UNSAT prediction and full trace accuracy for models trained and tested on different datasets in the training regime for number of variables $p \in [6, 10]$ and $p \in [11, 15]$. Columns denote train datasets, and rows denote test datasets. Each accuracy is computed over 10000 total samples.

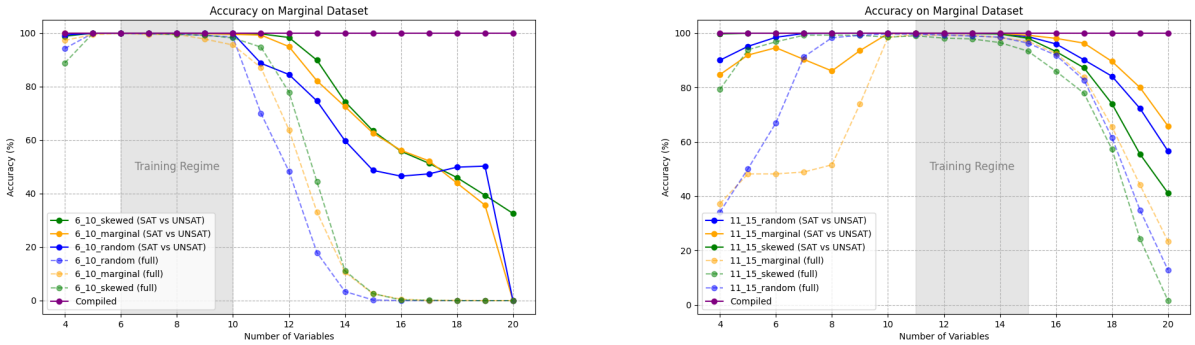

Figure 3. Result of the Length generalization experiments, showing SAT/UNSAT prediction accuracy (solid) and full trace accuracy (opaque, dashed) of Transformer models trained on the marginal, random, and skewed dataset with CoT on the marginal dataset over 4-20 variables. Left: model trained on 6-10 variables. Right: model trained on 11-15 variables. Compiled refers to the compiled model corresponding to our theoretical construction described in Section 5.1.

ated by the model adheres to the abstract DPLL procedure (Definition C.13) under our CoT definition. While strict, the "correct" CoT is not unique since the model may freely choose variable assignment and deduction orders.

6.1. Intra-length OOD Generalization

Our first set of experiments evaluates the model's performance on SAT formulas sampled from different distributions from training, but the number of variables in formulas remains the same ($p \in [6, 10]$ and $p \in [11, 15]$ for both train and test datasets).

As shown in Section 5.1, our trained models achieve near-perfect SAT vs UNSAT prediction accuracy when tested on the same number of variables as the training data, even when on formulas sampled from different distributions. The model also strictly follows a correct reasoning procedure for most samples. Recall that the "marginal" dataset has SAT vs UNSAT samples differing by a single token (out of at least $16p$ tokens in the input formula), which minimizes statistical evidence that can be used for SAT/UNSAT prediction. Our experiments suggest that the LLM have very likely learned logical reasoning procedures using CoT that can be applied to all formulas with the same number of variables as the data they are trained on.

6.2. Limitations in Length Generalization

The second experiment evaluates the model's ability to generalize to formulas with a different number of variables than seen during training. We use the model trained on 3 data distributions described in section 6.1 and evaluate the marginal dataset with 4-20 variables, with 2,000 samples each. For this experiment, we evaluate the accuracy of the binary SAT vs UNSAT prediction.

Results In Figure 3, our results indicate that performance degrades drastically beyond the training regime when the number of variables increases. This shows that the model is unable to learn a general SAT-solving algorithm that works for all inputs of arbitrary lengths, which corroborates our theoretical result where the size of the Transformer for SAT-solving depends on the number of variables.

Acknowledgements

This material is based upon work supported by the National Science Foundation under grant no. 2229876 and is supported in part by funds provided by the National Science Foundation, by the Department of Homeland Security, and by IBM. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation or its federal agency and industry partners. This work is supported by NSF award CCF-2403391. The authors thank Hassan Naveed for his exploratory work on using RASP for SAT-solver implementation, William Armstrong for suggesting the abstract DPLL paper and contributing code, and Jintong Jiang for providing suggestions on the paper's presentation and writing.

Impact Statement

This paper presents work whose goal is to advance the field of Machine Learning. There are many potential societal consequences of our work, none of which we feel must be specifically highlighted here

References

Biere, A., Heule, M., van Maaren, H., and Walsh, T. (eds.). *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*. IOS Press, 2009. ISBN 978-1-58603-929-5. URL http://dblp.uni-trier.de/db/series/faia/faia185.html.

Chen, M., Li, G., Wu, L.-I., Liu, R., Su, Y., Chang, X., and Xue, J. Can language models pretend solvers? logic code simulation with llms. In *International Symposium on Dependable Software Engineering: Theories, Tools, and Applications*, pp. 102–121. Springer, 2024.

Clark, P., Tafjord, O., and Richardson, K. Transformers as soft reasoners over language. In *Proceedings of the Twenty-Ninth International Joint Conference on Artificial Intelligence*, IJCAI'20, 2021. ISBN 9780999241165.

Cook, S. A. The complexity of theorem-proving procedures. In Harrison, M. A., Banerji, R. B., and Ullman, J. D. (eds.), *Proceedings of the 3rd Annual ACM Symposium on Theory of Computing, May 3-5, 1971, Shaker Heights, Ohio, USA*, pp. 151–158. ACM, 1971. doi: 10.1145/800157.805047. URL https://doi.org/10.1145/800157.805047.

Crawford, J. M. and Auton, L. D. Experimental results on the crossover point in random 3-sat. *Artificial Intelligence*, 81(1):31–57, 1996. ISSN 0004-3702. doi: https://doi.org/10.1016/0004-3702(95)00046-1. URL https://www.sciencedirect.com/science/article/pii/0004370295000461. Frontiers in Problem Solving: Phase Transitions and Complexity.

DeepSeek-AI, Guo, D., Yang, D., Zhang, H., Song, J., Zhang, R., Xu, R., Zhu, Q., Ma, S., Wang, P., Bi, X., Zhang, X., Yu, X., Wu, Y., Wu, Z. F., Gou, Z., Shao, Z., Li, Z., Gao, Z., Liu, A., Xue, B., Wang, B., Wu, B., Feng, B., Lu, C., Zhao, C., Deng, C., Zhang, C., Ruan, C., Dai, D., Chen, D., Ji, D., Li, E., Lin, F., Dai, F., Luo, F., Hao, G., Chen, G., Li, G., Zhang, H., Bao, H., Xu, H., Wang, H., Ding, H., Xin, H., Gao, H., Qu, H., Li, H., Guo, J., Li, J., Wang, J., Chen, J., Yuan, J., Qiu, J., Li, J., Cai, J. L., Ni, J., Liang, J., Chen, J., Dong, K., Hu, K., Gao, K., Guan, K., Huang, K., Yu, K., Wang, L., Zhang, L., Zhao, L., Wang, L., Zhang, L., Xu, L., Xia, L., Zhang, M., Zhang, M., Tang, M., Li, M., Wang, M., Li, M., Tian, N., Huang, P., Zhang, P., Wang, Q., Chen, Q., Du, Q., Ge, R., Zhang, R., Pan, R., Wang, R., Chen, R. J., Jin, R. L., Chen, R., Lu, S., Zhou, S., Chen, S., Ye, S., Wang, S., Yu, S., Zhou, S., Pan, S., Li, S. S., Zhou, S., Wu, S., Ye, S., Yun, T., Pei, T., Sun, T., Wang, T., Zeng, W., Zhao, W., Liu, W., Liang, W., Gao, W., Yu, W., Zhang, W., Xiao, W. L., An, W., Liu, X., Wang, X., Chen, X., Nie, X., Cheng, X., Liu, X., Xie, X., Liu, X., Yang, X., Li, X., Su, X., Lin, X., Li, X. Q., Jin, X., Shen, X., Chen, X., Sun, X., Wang, X., Song, X., Zhou, X., Wang, X., Shan, X., Li, Y. K., Wang, Y. Q., Wei, Y. X., Zhang, Y., Xu, Y., Li, Y., Zhao, Y., Sun, Y., Wang, Y., Yu, Y., Zhang, Y., Shi, Y., Xiong, Y., He, Y., Piao, Y., Wang, Y., Tan, Y., Ma, Y., Liu, Y., Guo, Y., Ou, Y., Wang, Y., Gong, Y., Zou, Y., He, Y., Xiong, Y., Luo, Y., You, Y., Liu, Y., Zhou, Y., Zhu, Y. X., Xu, Y., Huang, Y., Li, Y., Zheng, Y., Zhu, Y., Ma, Y., Tang, Y., Zha, Y., Yan, Y., Ren, Z. Z., Ren, Z., Sha, Z., Fu, Z., Xu, Z., Xie, Z., Zhang, Z., Hao, Z., Ma, Z., Yan, Z., Wu, Z., Gu, Z., Zhu, Z., Liu, Z., Li, Z., Xie, Z., Song, Z., Pan, Z., Huang, Z., Xu, Z., Zhang, Z., and Zhang, Z. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning, 2025. URL https://arxiv.org/abs/2501.12948.

Elhage, N., Nanda, N., Olsson, C., Henighan, T., Joseph, N., Mann, B., Askell, A., Bai, Y., Chen, A., Conerly, T., DasSarma, N., Drain, D., Ganguli, D., Hatfield-Dodds, Z., Hernandez, D., Jones, A., Kernion, J., Lovitt, L., Ndousse, K., Amodei, D., Brown, T., Clark, J., Kaplan, J., McCandlish, S., and Olah, C. A mathematical framework for transformer circuits. *Transformer Circuits Thread*, 2021. https://transformer-circuits.pub/2021/framework/index.html.

Fan, L., Hua, W., Li, L., Ling, H., and Zhang, Y. NPHardEval: Dynamic benchmark on reasoning ability of large language models via complexity classes. In Ku, L.-W., Mar-

tins, A., and Srikumar, V. (eds.), *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 4092–4114, Bangkok, Thailand, August 2024. Association for Computational Linguistics. doi: 10.18653/v1/2024.acl-long.225. URL https://aclanthology.org/2024.acl-long.225/.

Feng, G., Zhang, B., Gu, Y., Ye, H., He, D., and Wang, L. Towards revealing the mystery behind chain of thought: A theoretical perspective. In Oh, A., Naumann, T., Globerson, A., Saenko, K., Hardt, M., and Levine, S. (eds.), *Advances in Neural Information Processing Systems 36: Annual Conference on Neural Information Processing Systems 2023, NeurIPS 2023, New Orleans, LA, USA, December 10 - 16, 2023*, 2023.

Giannou, A., Rajput, S., Sohn, J.-Y., Lee, K., Lee, J. D., and Papailiopoulos, D. Looped transformers as programmable computers. In Krause, A., Brunskill, E., Cho, K., Engelhardt, B., Sabato, S., and Scarlett, J. (eds.), *Proceedings of the 40th International Conference on Machine Learning*, volume 202 of *Proceedings of Machine Learning Research*, pp. 11398–11442. PMLR, 23–29 Jul 2023. URL https://proceedings.mlr.press/v202/giannou23a.html.

Han, S., Schoelkopf, H., Zhao, Y., Qi, Z., Riddell, M., Zhou, W., Coady, J., Peng, D., Qiao, Y., Benson, L., Sun, L., Wardle-Solano, A., Szabó, H., Zubova, E., Burtell, M., Fan, J., Liu, Y., Wong, B., Sailor, M., Ni, A., Nan, L., Kasai, J., Yu, T., Zhang, R., Fabbri, A., Kryscinski, W. M., Yavuz, S., Liu, Y., Lin, X. V., Joty, S., Zhou, Y., Xiong, C., Ying, R., Cohan, A., and Radev, D. FOLIO: Natural language reasoning with first-order logic. In Al-Onaizan, Y., Bansal, M., and Chen, Y.-N. (eds.), *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, Miami, Florida, USA, November 2024. Association for Computational Linguistics. URL https://aclanthology.org/2024.emnlp-main.1229/.

Hazra, R., Venturato, G., Martires, P. Z. D., and Raedt, L. D. Can large language models reason? a characterization via 3-sat, 2024. URL https://arxiv.org/abs/2408.07215.

Jha, P., Jana, P., Suresh, P., Arora, A., and Ganesh, V. Rlsf: Reinforcement learning via symbolic feedback, 2024. URL https://arxiv.org/abs/2405.16661.

Kambhampati, S., Valmeekam, K., Guan, L., Verma, M., Stechly, K., Bhambri, S., Saldyt, L. P., and Murthy, A. B. Position: LLMs can't plan, but can help planning in LLM-modulo frameworks. In *Forty-first International Conference on Machine Learning*, 2024. URL https://openreview.net/forum?id=Th8JPEmH4z.

Li, Z., Liu, H., Zhou, D., and Ma, T. Chain of thought empowers transformers to solve inherently serial problems. In *The Twelfth International Conference on Learning Representations*, 2024. URL https://openreview.net/forum?id=3EWTEy9MTM.

Lindner, D., Kramar, J., Farquhar, S., Rahtz, M., McGrath, T., and Mikulik, V. Tracr: Compiled transformers as a laboratory for interpretability. In Oh, A., Naumann, T., Globerson, A., Saenko, K., Hardt, M., and Levine, S. (eds.), *Advances in Neural Information Processing Systems*, volume 36, pp. 37876–37899. Curran Associates, Inc., 2023.

Liu, B., Ash, J. T., Goel, S., Krishnamurthy, A., and Zhang, C. Transformers learn shortcuts to automata. In *The Eleventh International Conference on Learning Representations*, 2023a. URL https://openreview.net/forum?id=De4FYqjFueZ.

Liu, C., Lu, S., Chen, W., Jiang, D., Svyatkovskiy, A., Fu, S., Sundaresan, N., and Duan, N. Code execution with pre-trained language models. In Rogers, A., Boyd-Graber, J., and Okazaki, N. (eds.), *Findings of the Association for Computational Linguistics: ACL 2023*, pp. 4984–4999, Toronto, Canada, July 2023b. Association for Computational Linguistics. doi: 10.18653/v1/2023.findings-acl.308. URL https://aclanthology.org/2023.findings-acl.308/.

Merrill, W. and Sabharwal, A. 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.

Merrill, W. and Sabharwal, A. The expressive power of transformers with chain of thought. In *The Twelfth International Conference on Learning Representations*, 2024. URL https://openreview.net/forum?id=NjNGlPh8Wh.

Merrill, W., Sabharwal, A., and Smith, N. A. Saturated transformers are constant-depth threshold circuits. *Transactions of the Association for Computational Linguistics*, 10:843–856, 2021. URL https://api.semanticscholar.org/CorpusID:248085924.

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

Nieuwenhuis, R., Oliveras, A., and Tinelli, C. Abstract dpll and abstract dpll modulo theories. In Baader, F. and Voronkov, A. (eds.), *Logic for Programming, Artificial Intelligence, and Reasoning*, pp. 36–50, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg. ISBN 978-3-540-32275-7.

OpenAI. Openai o1 system card, 2024. URL https://cdn.openai.com/o1-system-card.pdf.

Pan, L., Albalak, A., Wang, X., and Wang, W. Logic-LM: Empowering large language models with symbolic solvers for faithful logical reasoning. In Bouamor, H., Pino, J., and Bali, K. (eds.), *Findings of the Association for Computational Linguistics: EMNLP 2023*, Singapore, December 2023. Association for Computational Linguistics. doi: 10.18653/v1/2023.findings-emnlp.248. URL https://aclanthology.org/2023.findings-emnlp.248/.

Paszke, A., Gross, S., Chintala, S., Chanan, G., Yang, E., DeVito, Z., Lin, Z., Desmaison, A., Antiga, L., and Lerer, A. Automatic differentiation in pytorch. 2017.

Pérez, J., Marinković, J., and Barceló, P. On the turing completeness of modern neural network architectures. In *International Conference on Learning Representations*, 2019. URL https://openreview.net/forum?id=HyGBdo0qFm.

Ryu, H., Kim, G., Lee, H. S., and Yang, E. Divide and translate: Compositional first-order logic translation and verification for complex logical reasoning. In *Proceedings of the 13th International Conference on Learning Representations (ICLR 2025)*. OpenReview, 2025. URL https://openreview.net/forum?id=09FiNmvNMw.

Saparov, A. and He, H. Language models are greedy reasoners: A systematic formal analysis of chain-of-thought. In *The Eleventh International Conference on Learning Representations*, 2023. URL https://openreview.net/forum?id=qFVVBzXxR2V.

Saparov, A., Pang, R. Y., Padmakumar, V., Joshi, N., Kazemi, S. M., Kim, N., and He, H. Testing the general deductive reasoning capacity of large language models using ood examples. In *Proceedings of the 37th International Conference on Neural Information Processing Systems*, NIPS '23, Red Hook, NY, USA, 2023. Curran Associates Inc.

Sato, T. and Kojima, R. Matsat: a matrix-based differentiable sat solver. *arXiv preprint arXiv:2108.06481*, 2021.

Selsam, D., Lamm, M., Bünz, B., Liang, P., de Moura, L., and Dill, D. L. Learning a sat solver from single-bit supervision. In *Proceedings of the 7th International Conference on Learning Representations (ICLR 2019)*, 2019. URL https://openreview.net/forum?id=HJMC_iA5tm.

Shi, Z., Li, M., Khan, S., Zhen, H.-L., Yuan, M., and Xu, Q. Satformer: Transformer-based unsat core learning. in 2023 ieee. In *ACM International Conference on Computer Aided Design (ICCAD)*, pp. 1–4, 2022.

Tafjord, O., Dalvi, B., and Clark, P. ProofWriter: Generating implications, proofs, and abductive statements over natural language. In Zong, C., Xia, F., Li, W., and Navigli, R. (eds.), *Findings of the Association for Computational Linguistics: ACL-IJCNLP 2021*, pp. 3621–3634, Online, August 2021. Association for Computational Linguistics. doi: 10.18653/v1/2021.findings-acl.317. URL https://aclanthology.org/2021.findings-acl.317/.

Touvron, H., Lavril, T., Izacard, G., Martinet, X., Lachaux, M.-A., Lacroix, T., Rozière, B., Goyal, N., Hambro, E., Azhar, F., Rodriguez, A., Joulin, A., Grave, E., and Lample, G. Llama: Open and efficient foundation language models. *ArXiv*, abs/2302.13971, 2023.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L., and Polosukhin, I. Attention is all you need. In Guyon, I., von Luxburg, U., Bengio, S., Wallach, H. M., Fergus, R., Vishwanathan, S. V. N., and Garnett, R. (eds.), *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, December 4-9, 2017, Long Beach, CA, USA*, pp. 5998–6008, 2017.

Wei, J., Wang, X., Schuurmans, D., Bosma, M., ichter, b., Xia, F., Chi, E., Le, Q. V., and Zhou, D. Chain-of-thought prompting elicits reasoning in large language models. In Koyejo, S., Mohamed, S., Agarwal, A., Belgrave, D., Cho, K., and Oh, A. (eds.), *Advances in Neural Information Processing Systems*, volume 35, pp. 24824–24837. Curran Associates, Inc., 2022.

Weiss, G., Goldberg, Y., and Yahav, E. Thinking like transformers. In Meila, M. and Zhang, T. (eds.), *Proceedings of the 38th International Conference on Machine Learning*, volume 139 of *Proceedings of Machine Learning Research*, pp. 11080–11090. PMLR, 18–24 Jul 2021. URL https://proceedings.mlr.press/v139/weiss21a.html.

Xu, J., Fei, H., Pan, L., Liu, Q., Lee, M.-L., and Hsu, W. Faithful logical reasoning via symbolic chain-of-thought. In Ku, L.-W., Martins, A., and Srikumar, V. (eds.), *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1:*

Long Papers), Bangkok, Thailand, August 2024. Association for Computational Linguistics. URL https://aclanthology.org/2024.acl-long.720/.

Yao, S., Peng, B., Papadimitriou, C., and Narasimhan, K. Self-attention networks can process bounded hierarchical languages. In *Association for Computational Linguistics (ACL)*, 2021.

Zhang, H., Li, L. H., Meng, T., Chang, K.-W., and Van Den Broeck, G. On the paradox of learning to reason from data. In *Proceedings of the Thirty-Second International Joint Conference on Artificial Intelligence*, IJCAI '23, 2023a. ISBN 978-1-956792-03-4. doi: 10.24963/ijcai.2023/375. URL https://doi.org/10.24963/ijcai.2023/375.

Zhang, H., Li, L. H., Meng, T., Chang, K.-W., and Van Den Broeck, G. On the paradox of learning to reason from data. In *Proceedings of the Thirty-Second International Joint Conference on Artificial Intelligence*, IJCAI '23, 2023b. ISBN 978-1-956792-03-4. doi: 10.24963/ijcai.2023/375. URL https://doi.org/10.24963/ijcai.2023/375.

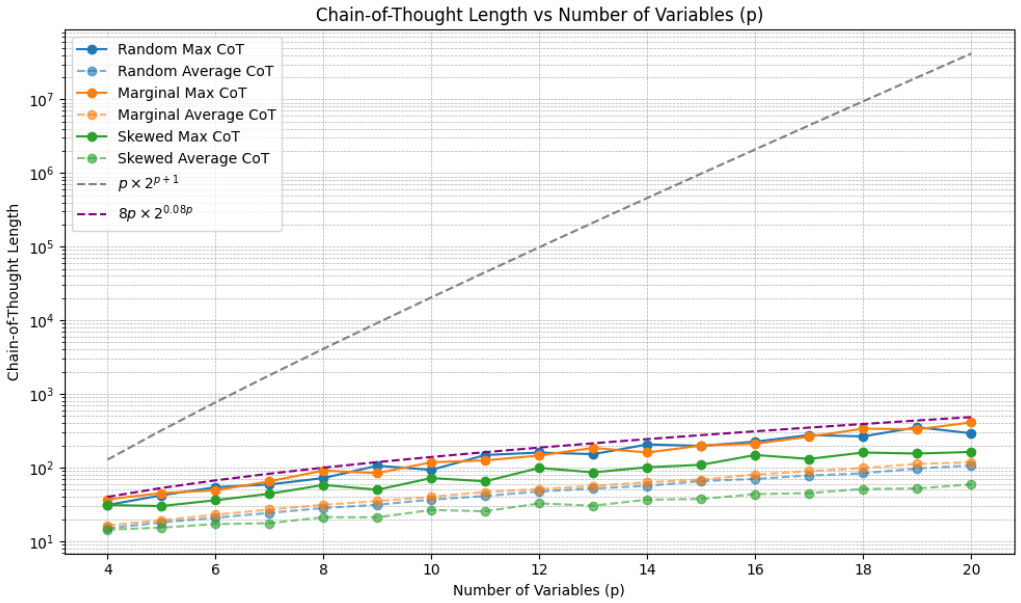

Figure 4. CoT Lengths generated by the compiled SAT-Solver Model vs the number of boolean variables in sampled SAT formulas, y-axis in log scale. Solid lines denote the maximum CoT length for each dataset while opaque, dashed lines denote the average CoT length. The empirical maximum CoT length in our datasets is bounded by $8p \cdot 2^{0.08p}$

.

A. Training Details

We use Llama (Touvron et al., 2023) models in the HuggingFace library. For the 70M model, we use models with 6 layers, 512 embedding dimensions, 8 heads, 512 attention hidden dimensions, and 2048 MLP hidden dimensions. For the 140M model, we use 12 layers, 768 embedding dimensions, 12 heads, 768 attention hidden dimensions, and 3072 MLP hidden dimensions. Both models have 850 context size. We trained for 5 epochs on both datasets using the Adam optimizer with a scheduled cosine learning rate decaying from 6×10^{-4} to 6×10^{-5} with $\beta_1 = 0.9$ and $\beta_2 = 0.95$.

B. Additional Experiment Results

Number of CoT Tokens for Theoretical Construction In Figure 4 we provide results on the number of CoT tokens required to solve randomly generated SAT instances. The number of CoT tokens required to decide 3-SAT instances are below our theoretical bound for all instances

Effect of Soft Attention In Figure 5 we provide results on how the SAT/UNSAT prediction accuracy is affected by numerical errors introduced by softmax. In particular, when the β value that controls the "hardness" of attention is below the necessary threshold, the resulting models can only achieve perfect accuracy for formulas with smaller sizes.

Length Generalization Results on Additional Datasets In Figure 6 we present results for length generalization (described in Section 6.2) on the marginal and skewed datasets.

C. Proofs

C.1. Preliminaries on Transformers Models

The Transformer architecture (Vaswani et al., 2017) is a foundational model in deep learning for sequence modeling tasks. In our work, we focus on the autoregressive decoder-only Transformer, which generates sequences by predicting the next token based on previously generated tokens. It is a relatively complex architecture, and here we only give a precise but quite concise description, and we refer the reader to (Vaswani et al., 2017), among many others, for additional details. Given an

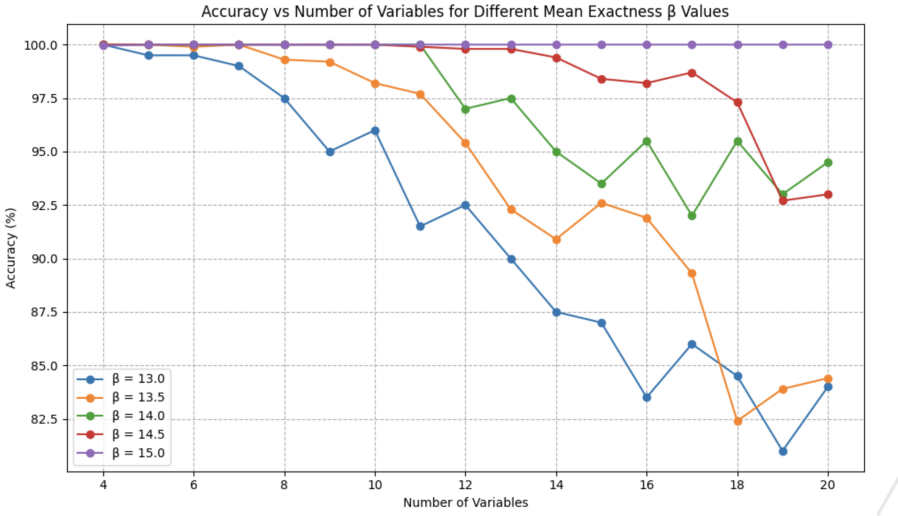

Figure 5. The impact of soft attention in Transformer layers on the SAT/UNSAT prediction accuracy. β is a scaling factor that allows the soft attention operation to better simulate hard attention at the cost of larger model parameter values in attention layers. The model achieves perfect accuracy on all "marginal" datasets starting at $\beta = 15$, and for lower β values, the resulting models can achieve perfect accuracy for formulas with variables of smaller sizes.

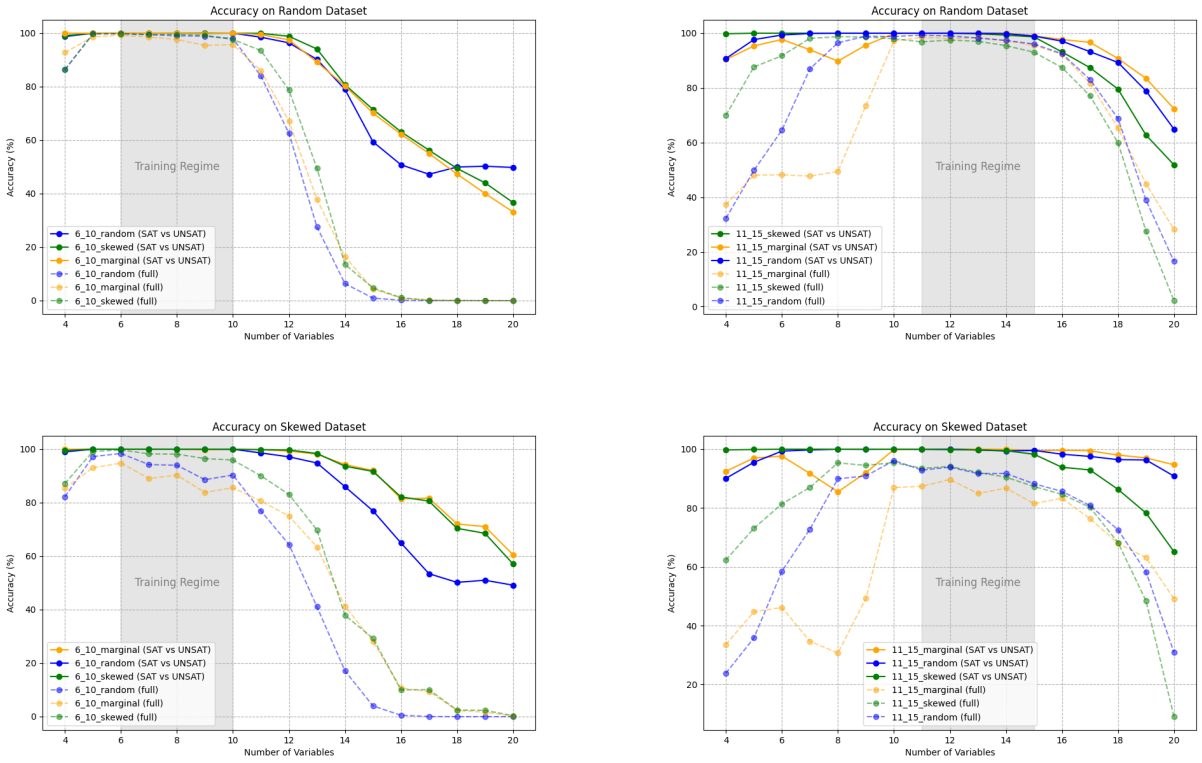

Figure 6. Result of the Length generalization experiments on the random and skewed evaluation dataset. The meaning of different lines are the same as Figure 3

input sequence of tokens $\mathbf{s} = (s_1, s_2, \ldots, s_n) \in \mathcal{V}^n$, where $\mathcal{V}$ is a *vocabulary*, a Transformer model $M : \mathcal{V}^* \to \mathcal{V}$ maps $\mathbf{s}$ to an output token $s_{n+1} \in \mathcal{V}$ by composing a sequence of parameterized intermediate operations. These begin with a token

embedding layer, following by L *transformer blocks* (*layers*), each block consisting of H *attention heads*, with embedding dimension d_{emb}, head dimension d_h, and MLP hidden dimension d_{mlp}. Let us now describe each of these maps in detail.

Token Embedding and Positional Encoding. Each input token s_i is converted into a continuous vector representation $\text{Embed}(s_i) \in \mathbb{R}^d$ using a fixed embedding map $\text{Emb}(\cdot)$. To incorporate positional information, a positional encoding vector $\boldsymbol{p}_i \in \mathbb{R}^d$ is added to each token embedding. The initial input to the first Transformer block is

$$\boldsymbol{X}^{(0)} \leftarrow (\text{Emb}(s_1) + \boldsymbol{p}_1, \ldots, \text{Emb}(s_n) + \boldsymbol{p}_n) \in \mathbb{R}^{n \times d}.$$

Transformer Blocks. For $l = 1, \ldots, L$, each block l of the transformer processes an embedded sequence $\boldsymbol{X}^{(l-1)} \in \mathbb{R}^{n \times d}$ to produce another embedded sequence $\boldsymbol{X}^{(l)} \in \mathbb{R}^{n \times d}$. Each block consists of a multi-head self-attention (MHA) mechanism and a position-wise feed-forward network (MLP). We have a set of parameter tensors that includes MLP parameters $\boldsymbol{W}_1^{(l)} \in \mathbb{R}^{d_{\text{emb}} \times d^*_{\text{mlp}}}$, $\boldsymbol{b}_1^{(l)} \in \mathbb{R}^{d^*_{\text{mlp}}}$, $\boldsymbol{W}_2^{(l)} \in \mathbb{R}^{d_{\text{mlp}} \times d}$, and $\boldsymbol{b}_2^{(l)} \in \mathbb{R}^d$, self-attention parameters $\boldsymbol{W}_Q^{(l,h)}$, $\boldsymbol{W}_K^{(l,h)}$, $\boldsymbol{W}_V^{(l,h)} \in \mathbb{R}^{d \times d_h}$ for every $h = 1, \ldots, H$, and multi-head projection matrix $\boldsymbol{W}_O^{(l)} \in \mathbb{R}^{(Hd_h) \times d_{\text{emb}}}$. We will collectively refer to all such parameters at layer l as $\Gamma^{(l)}$, whereas the self-attention parameters for attention head h at layer l will be referred to as $\Gamma^{(l,h)}$. We can now process the embedded sequence $\boldsymbol{X}^{(l-1)}$ to obtain $\boldsymbol{X}^{(l)}$ in two stages:

$$\boldsymbol{H}^{(l)} \leftarrow \boldsymbol{X}^{(l-1)} + \text{MHA}\left(\boldsymbol{X}^{(l-1)}; \Gamma^{(l)}\right)$$
$$\boldsymbol{X}^{(l)} \leftarrow \boldsymbol{H}^{(l)} + \text{MLP}\left(\boldsymbol{H}^{(l)}; \Gamma^{(l)}\right),$$

where

$$\text{MHA}\left(\boldsymbol{X}; \Gamma^{(l)}\right) := \left[\text{Att}(\boldsymbol{X}; \Gamma^{(l,1)}); \ldots; \text{Att}(\boldsymbol{X}; \Gamma^{(l,H)})\right] \boldsymbol{W}_O^{(l)}$$
$$\text{Att}(\boldsymbol{X}; \Gamma^{(l,h)}) := \sigma\left(\frac{\boldsymbol{X}\boldsymbol{W}_Q^{(l,h)}(\boldsymbol{W}_K^{(l,h)}\boldsymbol{X})^\top}{\sqrt{d_h}} + \boldsymbol{M}\right)\boldsymbol{X}\boldsymbol{W}_V^{(l,h)}$$
$$\text{MLP}\left(\boldsymbol{H}; \Gamma^{(l)}\right) := \text{act}\left(\boldsymbol{H}\boldsymbol{W}_1^{(l)} + \boldsymbol{b}_1^{(l)}\right)\boldsymbol{W}_2^{(l)} + \boldsymbol{b}_2^{(l)}.$$

The $n \times n$ matrix $\boldsymbol{M}$ is used as a "mask" to ensure self-attention is only backward-looking, so we set $\boldsymbol{M}[i,j] = -\infty$ for $i \geq j$ and $\boldsymbol{M}[i,j] = 0$ otherwise. σ represents the softmax operation. We use the $\text{ReGLU}(\cdot) : \mathbb{R}^{2d_{\text{mlp}}} \to \mathbb{R}^{d_{\text{mlp}}}$ activation function $\text{act}(\cdot)$ at each position. Given input $\boldsymbol{u} \in \mathbb{R}^{n \times 2d_{\text{mlp}}}$, for each position i we split $\boldsymbol{u}_i$ into two halves $\boldsymbol{u}_{i,1},\ \boldsymbol{u}_{i,2} \in \mathbb{R}^d$ and, using $\otimes$ denotes element-wise multiplication, we define

$$\sigma_{\text{ReGLU}}(\boldsymbol{u}_i) = \boldsymbol{u}_{i,1} \otimes \text{ReLU}(\boldsymbol{u}_{i,2}). \tag{1}$$

Output Layer. After the final Transformer block, the output representations are projected onto the vocabulary space to obtain a score for each token. We assume that we're using the greedy decoding strategy, where the token with the highest score at the last input position is the model output.

$$\boldsymbol{o} = \boldsymbol{X}^{(L)}\boldsymbol{W}_{\text{out}} + \boldsymbol{b}_{\text{out}} \in \mathbb{R}^{n \times V}, s_{n+1} = \arg\max_v \boldsymbol{o}_{n,v} \in \mathcal{V}$$

where $\boldsymbol{W}_{\text{out}} \in \mathbb{R}^{d \times V}$, $\boldsymbol{b}_{\text{out}} \in \mathbb{R}^V$, V is the size of the vocabulary, $\boldsymbol{o}_{n,v}$ is the score for token v at the last input position n.

Autoregressive Decoding and Chain-of-Thought. During generation, the Transformer model is repeatedly invoked to generate the next token and appended to the input tokens, described in Algorithm 1. In this paper, we refer to the full generated sequence of tokens as the **Chain-of-Thought (CoT)**, and the number of chain-of-thought tokens in Algorithm 1 is $t - n$.

Algorithm 1 Greedy Decoding

Require: Model $M : \mathcal{V}^* \to \mathcal{V}$, stop tokens $\mathcal{E} \subseteq \mathcal{V}$, prompt $\boldsymbol{s}_{1:n} = (s_1, s_2, \ldots, s_n)$, $t \leftarrow n$
1: **while** $t \leftarrow t + 1$ **do**
2: $\quad s_t \leftarrow M(\boldsymbol{s}_{1:t-1})$
3: $\quad$ **if** $s_t \in \mathcal{E}$ **return** $\boldsymbol{s}_{1:t}$
4: **end while**

C.2. 3-SAT

SAT The Boolean satisfiability problem (SAT) is the problem of determining whether there exists an assignment A of the variables in a Boolean formula F such that F is true under A.

3-SAT In this paper, we only consider 3-SAT instances in *conjunctive normal form* (CNF), where groups of at most 3 variables and their negations (*literals*) can be joined by OR operators into clauses, and these clauses can then be joined by AND operators. We use the well-known *DIMACS* encoding for CNF formulas where each literal is converted to a positive or negative integer corresponding to its index, and clauses are separated by a 0 (which represents an $\wedge$ operation). SAT problems where the Boolean formula is expressed in conjunctive normal form (CNF) with three literals per clause will be referred to as *3-SAT*. A formula in CNF is a conjunction (i.e. "AND") of clauses, a **clause** is a disjunction (i.e. "OR") of several **literals**, and each literal is either a variable or its negation. In the case of 3-SAT, each clause contains at most three literals. An example 3-SAT formula with 4 variables and 6 clauses is:

$$(x_1 \vee \neg x_2) \wedge (\neg x_1 \vee x_2 \vee \neg x_3) \wedge (x_2 \vee x_4 \vee \neg x_1) \wedge$$
$$(x_1 \vee \neg x_3 \vee x_4) \wedge (\neg x_2 \vee \neg x_3 \vee \neg x_4) \wedge (\neg x_4 \vee \neg x_1)$$

In the above formula, $(x_1 \vee \neg x_2)$ is a clause, which contains the literals x_1 and $\neg x_2$.

The 3-SAT problem refers to determining if any assignment of truth values to the variables allows the formula ϕ to evaluate as true. It is well-known that 3-SAT is NP-hard and is widely believed to be unsolvable in polynomial time.

DIMACS Encoding The DIMACS format is a standardized encoding scheme for representing Boolean formulas in conjunctive normal form (CNF) for SAT problems. Each clause in the formula is represented as a sequence of integers followed by a terminating "0" (i.e. "0" represents $\wedge$ symbols and parentheses). Positive integers correspond to variables, while negative integers represent the negations of variables. For instance, if a clause includes the literals x_1, $\neg x_2$, and x_3, it would be represented as "`1 -2 3 0`" in the DIMACS format.

For the 3-SAT example in the previous paragraph, the corresponding DIMACS representation would be:

```
1 -2 0 -1 2 -3 0 2 4 -1 0 1 -3 4 0 -2 -3 -4 0 -4 -1 0
```

Reducing a Formula. Let

$$F = \bigwedge_{i=1}^{c} C_i$$

be a 3-SAT formula, where each C_i is a clause (i.e. a disjunction of up to three literals). The *reduction* of F by A, denoted $F|_A$, is defined by:

1. **Remove (drop) any clause satisfied by A.**
   A clause C_i is satisfied by A if there is a literal $\ell \in C_i$ such that $\ell \in A$. In that case, C_i is automatically `True` and can be omitted from the conjunction.
2. **Delete (false) literals contradicting A.**
   For each remaining clause C_i, if it contains a literal ℓ that is *false* under A, remove that literal from C_i. Specifically:
   - If $x_j \in A$ (so x_j is `True`), then any literal $\neg x_j$ in C_i becomes false and is removed.
   - If $\neg x_j \in A$ (so x_j is `False`), then any literal x_j in C_i is removed.

   If a clause loses all its literals through this process, it becomes an *empty clause* and the formula is immediately `False`.

Formally, for each clause $C_i \subseteq L$, define

$$C_i|_A := \big(C_i \setminus \{\ell \in C_i : \ell \text{ is forced false by } A\}\big)$$

and keep $C_i|_A$ only if it is not already satisfied by A. Then

$$F|_A = \bigwedge_{\substack{i=1 \\ C_i \text{ not satisfied}}}^{c} \big(C_i|_A\big).$$

As an example, suppose

$$F \;=\; (x_1 \vee \neg x_2) \;\wedge\; (\neg x_1 \vee x_3) \;\wedge\; (x_2 \vee \neg x_3).$$

Let $A = \{x_1\}$. Then:

1. The first clause $(x_1 \vee \neg x_2)$ is satisfied by $x_1 \in A$. Hence we *remove* it from the formula.
2. In the second clause $(\neg x_1 \vee x_3)$, the literal $\neg x_1$ is false (since x_1 is set `True`). We remove $\neg x_1$ and are left with (x_3).
3. The third clause $(x_2 \vee \neg x_3)$ is untouched: x_1 does not appear, so no literal is removed. However, it is not satisfied by x_1, so we keep it.

Thus,

$$F|_A \;=\; (x_3) \;\wedge\; (x_2 \vee \neg x_3).$$

If a partial assignment forces a clause to become empty, the whole formula becomes unsatisfiable under that assignment. For instance, with

$$F \;=\; (x_1 \vee x_2) \;\wedge\; (\neg x_1 \vee \neg x_2),$$

and a partial assignment $A = \{x_1, x_2\}$, we see:

- The first clause $(x_1 \vee x_2)$ is satisfied by $x_1 \in A$ and gets removed.
- In the second clause $(\neg x_1 \vee \neg x_2)$, both $\neg x_1$ and $\neg x_2$ contradict A, so both are removed. This leaves the second clause empty, which means $F|_A$ is an empty conjunction (i.e. `False`).

Hence no full extension of A can satisfy F.

Unit Propagation. An additional reduction step performed in SAT solving is **unit propagation**. After applying a partial assignment A to a formula F (obtaining $F|_A$), some clauses may reduce to a *single literal* (called a *unit clause*). Formally, a clause $C = \{\ell_1, \ldots, \ell_k\}$ is **unit** if $k = 1$. If C is unit, its lone literal ℓ must be assigned `True` in any extension of A that satisfies F. Concretely:

1. **Identify unit clauses.** Scan the reduced formula $F|_A$. If there is a clause C_u with exactly one remaining literal ℓ, then ℓ is forced `True`.
2. **Extend the partial assignment.** Insert the forced literal ℓ into A.
3. **Re-reduce the formula.** Remove any clauses satisfied by ℓ, and remove $\neg\ell$ from all remaining clauses.

This process may uncover additional unit clauses in subsequent steps, so unit propagation continues iteratively until there are no more clauses of size 1. If at any point a clause becomes empty, we conclude that the current assignment A cannot be extended to a satisfying assignment.

Example. Consider $F = (x_1 \vee \neg x_2) \wedge (x_1 \vee x_3) \wedge (x_2 \vee \neg x_3)$ and a partial assignment $A = \{\neg x_1\}$.

- First, $F|_A$ removes x_1 (now falsified) from $(x_1 \vee x_3)$, leaving the unit clause (x_3). Thus x_3 is forced `True`.
- We add x_3 to A, giving $A \leftarrow A \cup \{x_3\}$. Re-reducing the formula removes any literal $\neg x_3$. If that step causes another clause to become unit, we repeat.

This iterative assignment of forced literals often simplifies the problem significantly before any broader search is required.

C.3. Proof of Lemma 4.7

We prove each of the three statements in the lemma, showing that the vector-based definitions correspond to the logical operations described.

1. Satisfiability Checking

$$A \models F \quad \Longleftrightarrow \quad \min_{i\in[c]} \Big(E(C_i) \cdot E(A) \Big) \geq 1.$$

Logical Interpretation. The left-hand side, $A \models F$, means that every clause C_i in F is satisfied by A. This is equivalent to saying that, for every clause C_i, there exists at least one literal $l \in C_i$ such that $l \in A$.

Vector Translation. For a clause C_i and a partial assignment A, the dot product $E(C_i) \cdot E(A)$ computes the number of literals in C_i that are also in A:

$$E(C_i) \cdot E(A) = \sum_{v=1}^{p} \mathbf{1}_{\{x_v \in C_i\}} \cdot \mathbf{1}_{\{x_v \in A\}} + \sum_{v=1}^{p} \mathbf{1}_{\{\neg x_v \in C_i\}} \cdot \mathbf{1}_{\{\neg x_v \in A\}} = |C_i \cap A|.$$

If $E(C_i) \cdot E(A) \geq 1$, this means there is at least one literal in $C_i \cap A$, and hence C_i is satisfied. Taking the minimum over all clauses ensures that every clause C_i is satisfied, which is precisely the condition for $A \models F$.

2. Conflict Detection

$$F \models \neg A \quad \Longleftrightarrow \quad \min_{i\in[c]} \Big(E(C_i) \cdot E_{\text{not-false}}(A) \Big) = 0.$$

Logical Interpretation. The left-hand side, $F \models \neg A$, means that F contradicts A, i.e., there exists a clause C_i in F such that all literals in C_i are forced false by A. This happens if and only if no literal in C_i is "not-false" under A.

Vector Translation. For a clause C_i, the dot product $E(C_i) \cdot E_{\text{not-false}}(A)$ computes the number of literals in C_i that are *not forced false* by A:

$$E(C_i) \cdot E_{\text{not-false}}(A) = \sum_{v=1}^{p} \mathbf{1}_{\{x_v \in C_i\}} \cdot \mathbf{1}_{\{\neg x_v \notin A\}} + \sum_{v=1}^{p} \mathbf{1}_{\{\neg x_v \in C_i\}} \cdot \mathbf{1}_{\{x_v \notin A\}}.$$

If $E(C_i) \cdot E_{\text{not-false}}(A) = 0$, this means all literals in C_i are forced false by A, and C_i is a contradiction. Taking the minimum over all clauses ensures that this happens for at least one clause, which corresponds to $F \models \neg A$.

3. Deduction (Unit Propagation)

$$E(D) \;=\; \max\Big(\min\Big(\sum_{i\in[c]} \mathbf{1}_{\{E(C_i)\cdot E_{\text{not-false}}(A)=1\}} \cdot E(C_i),\; 1 \Big) -\; E_{\text{assigned}}(A),\; 0 \Big).$$

Logical Interpretation. A clause C_i becomes a *unit clause* under A if all but one of its literals are forced false by A. In this case, the remaining literal must be set to `True` in any extension of A. The set D consists of all such literals deduced via unit propagation.

Vector Translation. For each clause C_i, the condition $E(C_i) \cdot E_{\text{not-false}}(A) = 1$ identifies unit clauses after reduction, i.e., those with exactly one literal not forced false by A. For such clauses, $E(C_i)$ encodes the remaining literal.

The summation

$$\sum_{i\in[c]} \mathbf{1}_{\{E(C_i)\cdot E_{\text{not-false}}(A)=1\}} \cdot E(C_i)$$

computes a vector where each coordinate accumulates contributions from unit clauses identifying the corresponding literal. Taking $\min(\cdot, 1)$ elementwise ensures that each coordinate is at most 1, avoiding overcounting. Finally, subtracting $E_{\text{assigned}}(A)$ removes literals that are already assigned by A, leaving only the newly deduced literals.

This matches the conditions for unit propagation as defined in Appendix C.2

□

C.4. Useful Lemmas for Transformers

In this section, several useful results on Transformer operations on their approximation capavilities. Specifically, an MLP with ReGLU can exactly simulate ReLU, linear operations, and multiplication without error. For Self-attention lemmas, we directly adapt from (Feng et al., 2023).

Lemmas for MLP with ReGLU activation This section shows several lemmas showing the capabilities of the self-attention operation and MLP layers to approximate high-level vector operations. These high-level operations are later used as building blocks for the Transformer SAT-solver. Specifically, with appropriate weight configurations, a 2-layer MLP with ReGLU activation $f(\boldsymbol{x}) = \boldsymbol{W}_2[(\boldsymbol{W}_1\boldsymbol{x} + \boldsymbol{b}) \otimes \mathrm{relu}(\boldsymbol{V}\boldsymbol{x} + \boldsymbol{c})]$ can approximate the following vector operations for arbitrary input $\boldsymbol{x}$:

- Simulate a 2-layer MLP with ReLU activation: $\boldsymbol{W}_2\,\mathrm{ReLU}(\boldsymbol{W}_1'\boldsymbol{x} + \boldsymbol{b}_1') + \boldsymbol{b}_2'$
- Simulate any linear operation $\boldsymbol{W}\boldsymbol{x}$
- Simulate element-wise multiplication: $\boldsymbol{x}_1 \otimes \boldsymbol{x}_2$

Lemma C.1 (Simulating a 2-Layer ReLU MLP with ReGLU Activation)**.** *A 2-layer MLP with ReGLU activation function can simulate any 2-layer MLP with ReLU activation function.*

Proof. Let the ReLU MLP be defined as:

$$g(\boldsymbol{x}) = \boldsymbol{W}_2'\,\mathrm{ReLU}(\boldsymbol{W}_1'\boldsymbol{x} + \boldsymbol{b}_1') + \boldsymbol{b}_2'.$$

Set the weights and biases of the ReGLU MLP as follows:

$$\begin{aligned} \boldsymbol{W}_1 &= \boldsymbol{0}, \quad \boldsymbol{b}_1 = \boldsymbol{1}, \\ \boldsymbol{V} &= \boldsymbol{W}_1', \quad \boldsymbol{b}_2 = \boldsymbol{b}_1', \\ \boldsymbol{W}_2 &= \boldsymbol{W}_2', \quad \boldsymbol{b} = \boldsymbol{b}_2'. \end{aligned}$$

Then, the ReGLU MLP computes:

$$f(\boldsymbol{x}) = \boldsymbol{W}_2'\left[(\boldsymbol{0}\cdot\boldsymbol{x} + \boldsymbol{1}) \otimes \mathrm{ReLU}(\boldsymbol{W}_1'\boldsymbol{x} + \boldsymbol{b}_1')\right] + \boldsymbol{b}_2'.$$

Simplifying:

$$f(\boldsymbol{x}) = \boldsymbol{W}_2'\left[\boldsymbol{1} \otimes \mathrm{ReLU}(\boldsymbol{W}_1'\boldsymbol{x} + \boldsymbol{b}_1')\right] + \boldsymbol{b}_2' = \boldsymbol{W}_2'\,\mathrm{ReLU}(\boldsymbol{W}_1'\boldsymbol{x} + \boldsymbol{b}_1') + \boldsymbol{b}_2' = g(\boldsymbol{x}).$$

Thus, the ReGLU MLP computes the same function as the ReLU MLP. □

Lemma C.2 (Simulating Linear Operations with ReGLU MLP)**.** *A 2-layer MLP with ReGLU activation can simulate any linear operation $f(\boldsymbol{x}) = \boldsymbol{W}\boldsymbol{x} + \boldsymbol{b}$.*

Proof. To compute a linear function using the ReGLU MLP, we can set the activation to act as a scalar multiplier of one. Set the weights and biases as:

$$\begin{aligned} \boldsymbol{W}_1 &= \boldsymbol{W}, \quad \boldsymbol{b}_1 = \boldsymbol{b}, \\ \boldsymbol{V} &= \boldsymbol{0}, \quad \boldsymbol{b}_2 = \boldsymbol{1}, \\ \boldsymbol{W}_2 &= \boldsymbol{I}, \quad \boldsymbol{b} = \boldsymbol{0}. \end{aligned}$$

Here, $\boldsymbol{I}$ is the identity matrix.

Since $\boldsymbol{V}\boldsymbol{x} + \boldsymbol{b}_2 = \boldsymbol{b}_2 = \boldsymbol{1}$, we have:

$$\mathrm{ReLU}(\boldsymbol{V}\boldsymbol{x} + \boldsymbol{b}_2) = \mathrm{ReLU}(\boldsymbol{1}) = \boldsymbol{1}.$$

Then, the ReGLU MLP computes:

$$f(\boldsymbol{x}) = \boldsymbol{I}\left[(\boldsymbol{W}\boldsymbol{x} + \boldsymbol{b}) \otimes \boldsymbol{1}\right] = \boldsymbol{W}\boldsymbol{x} + \boldsymbol{b}.$$

Thus, any linear operation can be represented by appropriately setting $\boldsymbol{W}_1$, $\boldsymbol{b}_1$, and $\boldsymbol{W}_2$. □

Lemma C.3 (Element-wise Multiplication via ReGLU MLP)**.** *A 2-layer MLP with ReGLU activation can compute the element-wise multiplication of two input vectors $\boldsymbol{x}_1$ and $\boldsymbol{x}_2$, that is,*

$$f(\boldsymbol{x}) = \boldsymbol{x}_1 \otimes \boldsymbol{x}_2,$$

where $\boldsymbol{x} = [\boldsymbol{x}_1; \boldsymbol{x}_2]$ denotes the concatenation of $\boldsymbol{x}_1$ and $\boldsymbol{x}_2$.

Proof. Let $\boldsymbol{x} = [\boldsymbol{x}_1; \boldsymbol{x}_2] \in \mathbb{R}^{2n}$, where $\boldsymbol{x}_1, \boldsymbol{x}_2 \in \mathbb{R}^n$.

Set the weights and biases:

$$\begin{aligned}
\boldsymbol{W}_1 &= \begin{bmatrix} \boldsymbol{I}_n \\ \boldsymbol{I}_n \end{bmatrix}, & \boldsymbol{b}_1 &= \boldsymbol{0}_{2n}, \\
\boldsymbol{V} &= \begin{bmatrix} \boldsymbol{I}_n \\ -\boldsymbol{I}_n \end{bmatrix}, & \boldsymbol{b}_2 &= \boldsymbol{0}_{2n}, \\
\boldsymbol{W}_2 &= \begin{bmatrix} \boldsymbol{I}_n & -\boldsymbol{I}_n \end{bmatrix}, & \boldsymbol{b} &= \boldsymbol{0}_n.
\end{aligned}$$

Compute:

$$\begin{aligned}
\boldsymbol{W}_1\boldsymbol{x} + \boldsymbol{b}_1 &= \begin{bmatrix} \boldsymbol{x}_1 \\ \boldsymbol{x}_1 \end{bmatrix}, \\
\boldsymbol{V}\boldsymbol{x} + \boldsymbol{b}_2 &= \begin{bmatrix} \boldsymbol{x}_2 \\ -\boldsymbol{x}_2 \end{bmatrix}, \\
\mathrm{ReLU}(\boldsymbol{V}\boldsymbol{x} + \boldsymbol{b}_2) &= \begin{bmatrix} \mathrm{ReLU}(\boldsymbol{x}_2) \\ \mathrm{ReLU}(-\boldsymbol{x}_2) \end{bmatrix}.
\end{aligned}$$

The element-wise product:

$$(\boldsymbol{W}_1\boldsymbol{x} + \boldsymbol{b}_1) \otimes \mathrm{ReLU}(\boldsymbol{V}\boldsymbol{x} + \boldsymbol{b}_2) = \begin{bmatrix} \boldsymbol{x}_1 \otimes \mathrm{ReLU}(\boldsymbol{x}_2) \\ \boldsymbol{x}_1 \otimes \mathrm{ReLU}(-\boldsymbol{x}_2) \end{bmatrix}.$$

Compute the output:

$$\begin{aligned}
f(\boldsymbol{x}) &= \boldsymbol{W}_2 \left[(\boldsymbol{W}_1\boldsymbol{x} + \boldsymbol{b}_1) \otimes \mathrm{ReLU}(\boldsymbol{V}\boldsymbol{x} + \boldsymbol{b}_2) \right] + \boldsymbol{b} \\
&= \boldsymbol{x}_1 \otimes \mathrm{ReLU}(\boldsymbol{x}_2) - \boldsymbol{x}_1 \otimes \mathrm{ReLU}(-\boldsymbol{x}_2) \\
&= \boldsymbol{x}_1 \otimes (\mathrm{ReLU}(\boldsymbol{x}_2) - \mathrm{ReLU}(-\boldsymbol{x}_2)) \\
&= \boldsymbol{x}_1 \otimes \boldsymbol{x}_2.
\end{aligned}$$

Thus, the ReGLU MLP computes $f(\boldsymbol{x}) = \boldsymbol{x}_1 \otimes \boldsymbol{x}_2$ without restrictions on $\boldsymbol{x}_2$. □

Capabilities of the Self-Attention Layer In this subsection, we provide 2 core lemmas on the capabilities of the self-attention layer from (Feng et al., 2023).

Let $n \in \mathbb{N}$ be an integer and let $\boldsymbol{x}_1, \boldsymbol{x}_2, \cdots, \boldsymbol{x}_n$ be a sequence of vectors where $\boldsymbol{x}_i = (\tilde{\boldsymbol{x}}_i, r_i, 1) \in [-M, M]^{d+2}$, $\tilde{\boldsymbol{x}}_i \in \mathbb{R}^d$, $r_i \in \mathbb{R}$, and M is a large constant. Let $\boldsymbol{K}, \boldsymbol{Q}, \boldsymbol{V} \in \mathbb{R}^{d' \times (d+2)}$ be any matrices with $\|\boldsymbol{V}\|_\infty \leq 1$, and let $0 < \rho, \delta < M$ be any real numbers. Denote $\boldsymbol{q}_i = \boldsymbol{Q}\boldsymbol{x}_i$, $\boldsymbol{k}_j = \boldsymbol{K}\boldsymbol{x}_j$, $\boldsymbol{v}_j = \boldsymbol{V}\boldsymbol{x}_j$, and define the *matching set* $\mathcal{S}_i = \{j \leq i : |\boldsymbol{q}_i \cdot \boldsymbol{k}_j| \leq \rho\}$. Equipped with these notations, we define two basic operations as follows:

- COPY: The output is a sequence of vectors $\boldsymbol{u}_1, \cdots, \boldsymbol{u}_n$ with $\boldsymbol{u}_i = \boldsymbol{v}_{\mathrm{pos}(i)}$, where $\mathrm{pos}(i) = \mathrm{argmax}_{j \in \mathcal{S}_i} r_j$.
- MEAN: The output is a sequence of vectors $\boldsymbol{u}_1, \cdots, \boldsymbol{u}_n$ with $\boldsymbol{u}_i = \mathrm{mean}_{j \in \mathcal{S}_i} \boldsymbol{v}_j$.

Assumption C.4. [Assumption C.6 from (Feng et al., 2023)] The matrices $\boldsymbol{Q}, \boldsymbol{K}, \boldsymbol{V}$ and scalars ρ, δ satisfy that for all considered sequences $\boldsymbol{x}_1, \boldsymbol{x}_2, \cdots, \boldsymbol{x}_n$, the following hold:

- For any $i, j \in [n]$, either $|\boldsymbol{q}_i \cdot \boldsymbol{k}_j| \leq \rho$ or $\boldsymbol{q}_i \cdot \boldsymbol{k}_j \leq -\delta$.

- For any $i, j \in [n]$, either $i = j$ or $|r_i - r_j| \geq \delta$.

Assumption C.4 says that there are sufficient gaps between the attended position (e.g., $\mathrm{pos}(i)$) and other positions. The two lemmas below show that the attention layer with casual mask can implement both COPY operation and MEAN operation efficiently.

Lemma C.5 (Lemma C.7 from (Feng et al., 2023)). *Assume Assumption C.4 holds with $\rho \leq \frac{\delta^2}{8M}$. For any $\epsilon > 0$, there exists an attention layer with embedding size $O(d)$ and one causal attention head that can approximate the COPY operation defined above. Formally, for any considered sequence of vectors $\boldsymbol{x}_1, \boldsymbol{x}_2, \ldots, \boldsymbol{x}_n$, denote the corresponding attention output as $\boldsymbol{o}_1, \boldsymbol{o}_2, \ldots, \boldsymbol{o}_n$. Then, we have $\|\boldsymbol{o}_i - \boldsymbol{u}_i\|_\infty \leq \epsilon$ for all $i \in [n]$ with $\mathcal{S}_i \neq \emptyset$. Moreover, the ℓ_∞ norm of attention parameters is bounded by $O(\mathrm{poly}(M, 1/\delta, \log(n), \log(1/\epsilon)))$.*

Lemma C.6 (Lemma C.8 from (Feng et al., 2023)). *Assume Assumption C.4 holds with $\rho \leq \frac{\delta\epsilon}{16M\ln(\frac{4Mn}{\epsilon})}$. For any $0 < \epsilon \leq M$, there exists an attention layer with embedding size $O(d)$ and one causal attention head that can approximate the MEAN operation defined above. Formally, for any considered sequence of vectors $\boldsymbol{x}_1, \boldsymbol{x}_2, \ldots, \boldsymbol{x}_n$, denote the attention output as $\boldsymbol{o}_1, \boldsymbol{o}_2, \ldots, \boldsymbol{o}_n$. Then, we have $\|\boldsymbol{o}_i - \boldsymbol{u}_i\|_\infty \leq \epsilon$ for all $i \in [n]$ with $\mathcal{S}_i \neq \emptyset$. Moreover, the ℓ_∞ norm of attention parameters is bounded by $O(\mathrm{poly}(M, 1/\delta, \log(n), \log(1/\epsilon)))$.*

C.5. Saturated Attention

To introduce our construction of Transformer layers and attention head, we first introduce *saturated self-attention*, which is an idealization of the usual softmax attention head that allows for sparse and uniform attention (i.e. "hard" attention):

Definition C.7 (Saturated Masked Attention, Merrill et al. (2022)). A saturated attention head with hidden dimension d_h, embedding dimension d_{emb} and weight $\Gamma_s = (\boldsymbol{W}_Q, \boldsymbol{W}_K, \boldsymbol{W}_V)$ is a function $\mathrm{SaturatedAttn}(\boldsymbol{X}; \Gamma_s) : \mathbb{R}^{n \times d_{\mathrm{emb}}} \to \mathbb{R}^{n \times d_{\mathrm{h}}}$ that satisfy the following:

$$\boldsymbol{A} := \boldsymbol{X}\boldsymbol{W}_Q(\boldsymbol{W}_K\boldsymbol{X})^\top \in \mathbb{R}^{n \times n}$$
$$\mathcal{M}_i := \{j \in [i] | \boldsymbol{A}_{ij} = \max_k \boldsymbol{A}_{ik}\}$$
$$\mathrm{SaturatedAttn}(\boldsymbol{X}; \Gamma_s)_i := \frac{\sum_{j \in \mathcal{M}_i} \boldsymbol{X}_j \boldsymbol{W}_V}{|\mathcal{M}_i|}$$

Intuitively, while softmax attention computes a distribution of attention over all previous positions and computes a weighted average, saturated attention only attends to the previous positions with the highest attention value and computes a uniform average over these positions.

We now show that Saturated Attention can be approximated by normal softmax attention:

Corollary C.8 (Softmax Attention Can Approximate Saturated Attention, implied by Lemma C.6). *Let $n \in \mathbb{N}$. Consider any input sequence $\boldsymbol{X} \in \mathbb{R}^{n \times d_{\mathrm{emb}}}$, and let $\mathrm{SaturatedAttn}(\boldsymbol{X}; \Gamma_s)$ be a saturated attention head with a causal mask and parameter norm bounded by $O(1)$ that produces outputs $\mathbf{o}_1, \ldots, \mathbf{o}_n \in \mathbb{R}^{d_h}$.*

Suppose further that, for each row i, the maximum attention score $\max_{j \leq i}(\boldsymbol{A}_{ij})$ of the saturated head exceeds all other scores by a margin of at least $\delta > 0$, i.e. if $j \in \mathcal{M}_i$ (the set of maximizing indices) and $k \notin \mathcal{M}_i$, then $\boldsymbol{A}_{ij} - \boldsymbol{A}_{ik} \geq \delta$.

Then for any $\varepsilon > 0$, there exists a standard single-head softmax attention *function $\mathrm{Attn}(\boldsymbol{X}; \Gamma)$ with parameter norms bounded by $\mathrm{poly}(M, 1/\delta, \log(n), \log(1/\varepsilon))$ such that its outputs $\tilde{\mathbf{o}}_1, \ldots, \tilde{\mathbf{o}}_n \in \mathbb{R}^{d_h}$ satisfy*

$$\left\|\tilde{\mathbf{o}}_i - \mathbf{o}_i\right\|_\infty \leq \varepsilon \quad \text{for all } 1 \leq i \leq n.$$

In other words, *if a saturated attention head has a strict dot-product margin among the top positions, it can be approximated arbitrarily closely by an ordinary causal softmax attention mechanism, using parameter magnitudes that grow at most polynomially in $1/\delta$, M, $\log(n)$, and $\log(1/\varepsilon)$.*

C.6. Proof of Lemma 4.8

We proof a version of Lemma 4.8 that uses saturated attention. Lemma 4.8 is immediately implied by the following lemma and Corollary C.8

Lemma C.9 (Saturated Masked Attention version of Lemma 4.8)**.** *Let F be a 3-SAT formula over variables $\{x_1, \ldots, x_p\}$ with c clauses $\{C_1, \ldots, C_c\}$ and A a partial assignment defined on variables $\{x_1, \ldots, x_p\}$. Let*

$$\boldsymbol{X}_{encoding} = \begin{bmatrix} 0 & 1 & 1 \\ E(C_1) & 0 & 1 \\ \vdots & \vdots & \vdots \\ E(C_c) & 0 & 1 \\ E(A) & 0 & 1 \end{bmatrix} \in \mathbb{R}^{(c+2)\times(2p+2)}$$

Then given X as input, there exists:

- *An saturated attention head with parameters $\Gamma_s^{A\models F}$ and hidden dimension 1 that satisfies*
$$\text{SaturatedAttn}(\boldsymbol{X}; \Gamma_s^{A\models F})_{c+2} = \mathbf{1}_{A\models F}$$
- *An saturated attention head with parameters $\Gamma_s^{F\models\neg A}$ and hidden dimension 1 that satisfies*
$$\text{SaturatedAttn}(\boldsymbol{X}; \Gamma_s^{F\models\neg A})_{c+2} = \mathbf{1}_{F\models\neg A}$$
- *An saturated attention head with parameters Γ_s^D with hidden dimension $2p$ and MLP layer with parameters Γ_{MLP}^D satisfy:*
$$MLP([\text{SaturatedAttn}(\boldsymbol{X}; \Gamma_s^D); \boldsymbol{X}]; \Gamma_{MLP}^D)_{c+2} = E(D)$$
unless $F \models \neg A$, where $E(D)$ is as defined in 4.7

Proof. We prove each of the three constructions in turn, using the definition of saturated attention (Definition C.7) and standard reductions from the logical semantics to dot-product comparisons.

We explain how to construct parameter matrices $(\boldsymbol{W}_Q, \boldsymbol{W}_K, \boldsymbol{W}_V)$ such that the resulting saturated attention head implements:

1. a check for $\mathbf{1}_{A\models F}$ (i.e. whether A satisfies F),
2. a check for $\mathbf{1}_{F\models\neg A}$ (i.e. whether A contradicts F),
3. a step of unit propagation that yields $E(D)$, provided $F \not\models \neg A$.

Within the following proof of Lemma C.9, we shorten $\boldsymbol{X}_{encoding}$ as $\boldsymbol{X}$.

1. Checking Satisfiability ($A \models F$)

We construct the matrices

$$\boldsymbol{W}_Q^{A\models F} \in \mathbb{R}^{(2p+2)\times(2p+1)}, \quad \boldsymbol{W}_K^{A\models F} \in \mathbb{R}^{(2p+2)\times(2p+1)}, \quad \boldsymbol{W}_V^{A\models F} \in \mathbb{R}^{(2p+2)\times 1}$$

as follows (with block-wise or coordinate-wise $\mathbf{0}$ and $\mathbf{0}_{2p}$ denoting matrices/vectors of all zeros of dimension $2p$ where the dimension subscript is omitted if they can be inferred from other entries, and $\boldsymbol{I}_{2p}$ the $2p \times 2p$ identity matrix).

$$\boldsymbol{W}_Q^{A\models F} = \begin{bmatrix} \boldsymbol{I}_{2p} & 0 \\ \mathbf{0}^\top & 0 \\ \mathbf{0}^\top & 1 \end{bmatrix} \qquad \boldsymbol{W}_K^{A\models F} = \begin{bmatrix} -\boldsymbol{I}_{2p} & 0 \\ \mathbf{0}^\top & -0.5 \\ \mathbf{0}^\top & 0 \end{bmatrix} \qquad \boldsymbol{W}_V^{A\models F} = \begin{bmatrix} \mathbf{0}_{2p} \\ 1 \\ 0 \end{bmatrix}.$$

Then

$$\boldsymbol{X}\boldsymbol{W}_Q^{A\models F} = \begin{bmatrix} \mathbf{0}_{2p} & 1 \\ E(C_1) & 1 \\ \vdots & \vdots \\ E(C_c) & 1 \\ E(A) & 1 \end{bmatrix} \quad \boldsymbol{X}\boldsymbol{W}_K^{A\models F} = \begin{bmatrix} \mathbf{0}_{2p} & -0.5 \\ -E(C_1) & 0 \\ \vdots & \vdots \\ -E(C_c) & 0 \\ -E(A) & 0 \end{bmatrix} \quad \boldsymbol{X}\boldsymbol{W}_V^{A\models F} = \begin{bmatrix} 1 \\ 0 \\ \vdots \\ 0 \end{bmatrix}$$

$$\boldsymbol{A} := \boldsymbol{X}\boldsymbol{W}_Q^{A\models F}(\boldsymbol{W}_K^{A\models F}\boldsymbol{X})^\top = \begin{bmatrix} -0.5 & 0 & 0 & \dots & 0 \\ -0.5 & -E(C_1)\cdot E(C_1) & -E(C_1)\cdot E(C_2) & \dots & -E(C_1)\cdot E(A) \\ -0.5 & -E(C_2)\cdot E(C_1) & -E(C_2)\cdot E(C_2) & \dots & -E(C_2)\cdot E(A) \\ \vdots & \vdots & \vdots & & \vdots \\ -0.5 & -E(A)\cdot E(C_1) & -E(A)\cdot E(C_2) & \dots & -E(A)\cdot E(A) \end{bmatrix}$$

Since we want to output $\mathbf{1}_{A\models F}$ at the last position $c+2$ corresponding to $E(A)$ in $\boldsymbol{X}_{encoding}$, we focus on the last row of A:

$$\boldsymbol{A}_{c+2} = [-0.5 \quad -E(A)\cdot E(C_1) \quad -E(A)\cdot E(C_2) \quad \dots \quad -E(A)\cdot E(C_c) \quad -E(A)\cdot E(A)]$$

Now consider $\mathcal{M}_{c+2} = \{j \in [c+2] | \boldsymbol{A}_{(c+2),j} = \max_k \boldsymbol{A}_{(c+2),k}\}$. Note that $\forall i \in [c], E(A)\cdot E(C_i) \in \mathbb{N}$ and since $\boldsymbol{A}_{(c+2),1} = -0.5$ there is:

$$\mathcal{M}_{c+2} = \{1\} \iff \min_{i\in[c]} E(C_i)\cdot E(A) \geq 1.$$

$$\mathcal{M}_{c+2} \subset [2, c+2] \iff \min_{i\in[c]} E(C_i)\cdot E(A) = 0.$$

which are the only 2 possibilities for nonnegative integers $E(C_i)\cdot E(A)$. Also, since $(\boldsymbol{X}\boldsymbol{W}_V^{A\models F})^\top = [1\ 0\ 0\ \dots\ 0]$ we have that

$$\boldsymbol{X}_j\boldsymbol{W}_V^{A\models F} = \begin{cases} 1 & \text{if } j = 1 \\ 0 & \text{otherwise} \end{cases}$$

$$\begin{aligned} \text{SaturatedAttn}(\boldsymbol{X}; \Gamma_s)_{c+2} &:= \frac{\sum_{j\in\mathcal{M}_{c+2}} \boldsymbol{X}_j\boldsymbol{W}_V^{A\models F}}{|\mathcal{M}_{c+2}|} \\ &= \begin{cases} \frac{1}{1} & \text{if } \mathcal{M}_{c+2} = \{1\} \\ 0 & \text{if } \mathcal{M}_{c+2} \subset [2, c+2] \end{cases} \\ &= \mathbf{1}_{\mathcal{M}_{c+2}=\{1\}} \\ &= \mathbf{1}_{\min_{i\in[c]} E(C_i)\cdot E(A)\geq 1} \\ &= \mathbf{1}_{A\models F} \end{aligned}$$

where the last step is by Lemma 4.7. This concludes our proof for satisfiability checking.

2. Detecting Conflict ($F \models \neg A$)

Note that for $B \in \mathcal{B}$ we have

$$E_{\text{not-false}}(B) = \begin{bmatrix} \mathbf{0}_{p\times p} & -\boldsymbol{I}_p \\ -\boldsymbol{I}_p & \mathbf{0}_{p\times p} \end{bmatrix} E(B) + \mathbf{1}_p$$

Define

$$\boldsymbol{P}_{\text{not-false}} := \begin{bmatrix} \mathbf{0}_{p\times p} & -\boldsymbol{I}_p & \mathbf{0}_p & \mathbf{0}_p \\ -\boldsymbol{I}_p & \mathbf{0}_{p\times p} & \mathbf{0}_p & \mathbf{0}_p \\ \mathbf{0}_p^\top & \mathbf{0}_p^\top & 1 & 0 \\ \mathbf{1}_p^\top & \mathbf{1}_p^\top & 0 & 1 \end{bmatrix} \in \mathbb{R}^{(2p+2)\times(2p+2)}$$

Then

$$\boldsymbol{X}\boldsymbol{P}_{\text{not-false}} = \begin{bmatrix} 0 & 1 & 1 \\ E_{\text{not-false}}(C_1) & 0 & 1 \\ \vdots & \vdots & \vdots \\ E_{\text{not-false}}(C_c) & 0 & 1 \\ E_{\text{not-false}}(A) & 0 & 1 \end{bmatrix} \in \mathbb{R}^{(c+2)\times(2p+2)}$$

We now construct the matrices

$$\boldsymbol{W}_Q^{F\models\neg A} \in \mathbb{R}^{(2p+2)\times(2p+1)}, \quad \boldsymbol{W}_K^{F\models\neg A} \in \mathbb{R}^{(2p+2)\times(2p+1)}, \quad \boldsymbol{W}_V^{F\models\neg A} \in \mathbb{R}^{(2p+2)\times 1}$$

as follows:

$$\boldsymbol{W}_Q^{F\models\neg A} = \boldsymbol{P}_{\text{not-false}}\boldsymbol{W}_Q^{A\models F} \qquad \boldsymbol{W}_K^{F\models\neg A} = \boldsymbol{W}_K^{A\models F} \qquad \boldsymbol{W}_V^{F\models\neg A} = \begin{bmatrix} \mathbf{0}_{2p} \\ -1 \\ 1 \end{bmatrix}.$$

Then

$$\boldsymbol{X}\boldsymbol{W}_Q^{F\models\neg A} = \begin{bmatrix} \mathbf{0}_{2p} & 1 \\ E_{\text{not-false}}(C_1) & 1 \\ \vdots & \vdots \\ E_{\text{not-false}}(C_c) & 1 \\ E_{\text{not-false}}(A) & 1 \end{bmatrix} \quad \boldsymbol{X}\boldsymbol{W}_K^{F\models\neg A} = \begin{bmatrix} \mathbf{0}_{2p} & -0.5 \\ -E(C_1) & 0 \\ \vdots & \vdots \\ -E(C_c) & 0 \\ -E(A) & 0 \end{bmatrix} \quad \boldsymbol{X}\boldsymbol{W}_V^{F\models\neg A} = \begin{bmatrix} 0 \\ 1 \\ \vdots \\ 1 \end{bmatrix}$$

Recall from Lemma 4.7 that:

$$F \models \neg A \quad \Longleftrightarrow \quad \min_{i\in[c]} \Big(E(C_i) \cdot E_{\text{not-false}}(A) \Big) = 0.$$

The remaining argument is very similar to satisfiability checking and we omit the full proof.

3. Unit Propagation (D)

Recall that $D := \{l \in L \mid F \wedge A \models_1 l\}$ and

$$E(D) = \max\Big[\min\Big(\sum_{i\in[c]} E(C_i)\mathbf{1}_{\{E(C_i)\cdot E_{\text{not-false}}(A)=1\}}, 1\Big) - E_{\text{assigned}}(A),\ 0\Big]. \tag{2}$$

To address unit propagation with saturated attention, we use a slightly different formulation than the formula in Lemma 4.7:

Proposition C.10. *Let $m > 1$ be an arbitrary constant, then*

$$\boldsymbol{z} := \sum_{i\in[c]} \mathbf{1}_{\{E(C_i)\cdot E_{\text{not-false}}(A)=1\}} \cdot E(C_i)$$
$$E(D) = \text{ReLU}(m\boldsymbol{z} - E_{\text{assigned}}(A)) - \text{ReLU}(m\boldsymbol{z} - 1)$$

Proof. We start from the expression in equation 2,

$$E(D) \;=\; \max\Big[\min\big(\boldsymbol{z},\,1\big) \;-\; E_{\text{assigned}}(A),\ 0\Big], \quad \text{where} \quad \boldsymbol{z} \;:=\; \sum_{i\in[c]} \mathbf{1}_{\{\,E(C_i)\cdot E_{\text{not-false}}(A)=1\}} \;\cdot\; E(C_i).$$

Because $E_{\text{assigned}}(A) \in \{0,1\}^{2p}$, each coordinate of $E_{\text{assigned}}(A)$ is either 0 or 1. A straightforward elementwise check shows the identity

$$\max\big(\min(a,1) \;-\; b,\,0\big) \;=\; \text{ReLU}\big(ma - b\big) \;-\; \text{ReLU}(ma - 1),$$

whenever $b \in \{0,1\}$. Indeed:

- If $b = 0$, then the left side is $\max(\min(a, 1), 0)$; on the right side,
$$\mathrm{ReLU}(ma) - \mathrm{ReLU}(ma - 1)$$
exactly matches $\max(\min(ma, 1), 0) = \max(\min(a, 1), 0)$ for any $a \geq 1$ (this is a standard piecewise identity).

- If $b = 1$, then $\min(a, 1) - 1 \leq 0$, hence the left side is always 0. On the right side,
$$\mathrm{ReLU}(ma - 1) \; - \; \mathrm{ReLU}(ma - 1) \; = \; 0.$$

Applying this identity coordinatewise, we obtain

$$\max\Big[\min(m\boldsymbol{z}, 1) \; - \; E_{\text{assigned}}(A), \; 0\Big] \; = \; \mathrm{ReLU}(m\boldsymbol{z} - E_{\text{assigned}}(A)) \; - \; \mathrm{ReLU}(m\boldsymbol{z} - 1),$$

which matches the stated expression for $E(D)$. □

We now construct the matrices

$$\boldsymbol{W}_Q^D \in \mathbb{R}^{(2p+2)\times(2p+1)}, \quad \boldsymbol{W}_K^D \in \mathbb{R}^{(2p+2)\times(2p+1)}, \quad \boldsymbol{W}_V^D \in \mathbb{R}^{(2p+2)\times(2p)}$$

as follows:

$$\boldsymbol{W}_Q^D \; = \; \boldsymbol{W}_Q^{F\models\neg A} \qquad \boldsymbol{W}_K^D \; = \; \begin{bmatrix} -\boldsymbol{I}_{2p} & 0 \\ \mathbf{0}^\top & -1.5 \\ \mathbf{0}^\top & 0 \end{bmatrix} \quad \boldsymbol{W}_V^D \; = \; c \begin{bmatrix} \boldsymbol{I}_p \\ \mathbf{0}_p^\top \\ \mathbf{0}_p^\top \end{bmatrix}.$$

Then

$$\boldsymbol{X}\boldsymbol{W}_Q^D = \begin{bmatrix} \mathbf{0}_{2p} & 1 \\ E_{\text{not-false}}(C_1) & 1 \\ \vdots & \vdots \\ E_{\text{not-false}}(C_c) & 1 \\ E_{\text{not-false}}(A) & 1 \end{bmatrix} \quad \boldsymbol{X}\boldsymbol{W}_K^D = \begin{bmatrix} \mathbf{0}_{2p} & -1.5 \\ -E(C_1) & 0 \\ \vdots & \vdots \\ -E(C_c) & 0 \\ -E(A) & 0 \end{bmatrix} \quad \boldsymbol{X}\boldsymbol{W}_V^D = c \begin{bmatrix} \mathbf{0}_p \\ E(C_1) \\ \vdots \\ E(A) \end{bmatrix}$$

We focus on the last row of $\boldsymbol{A} := \boldsymbol{X}\boldsymbol{W}_Q^D(\boldsymbol{W}_K^D\boldsymbol{X})^\top$:

$$\boldsymbol{A}_{c+2} = [-1.5 \quad -E(A)\cdot E_{\text{not-false}}(C_1) \quad -E(A)\cdot E_{\text{not-false}}(C_2) \quad \ldots \quad -E(A)\cdot E_{\text{not-false}}(C_c) \quad -E(A)\cdot E_{\text{not-false}}(A)]$$

Also, recall that we assume here $F \not\models \neg A$, so $\forall i, E(A)\cdot E_{\text{not-false}}(C_i) \geq 1$ and therefore $E(A)\cdot E_{\text{not-false}}(C_i)$ are positive integers. :

$$\mathcal{M}_{c+2} = \{1\} \quad \Longleftrightarrow \quad \min_{i\in[c]} E(C_i)\cdot E(A) \geq 2.$$

$$\mathcal{M}_{c+2} \subset [2, c+2] \quad \Longleftrightarrow \quad \min_{i\in[c]} E(C_i)\cdot E(A) = 1.$$

In particular:

$$\mathcal{M}_{c+2} = \begin{cases} \{1\} & \text{if } \min_{i\in[c]} E(C_i)\cdot E_{\text{assigned}}(A) \geq 2 \\ \{j \in [c] | E(C_i)\cdot E_{\text{assigned}}(A) = 1\} & \text{otherwise} \end{cases}$$

As a result:

$$\begin{aligned}\text{SaturatedAttn}(\boldsymbol{X};\Gamma_s)_{c+2} &:= \frac{\sum_{j\in\mathcal{M}_{c+2}}\boldsymbol{X}_j\boldsymbol{W}_V^D}{|\mathcal{M}_{c+2}|}\\ &= \begin{cases}\mathbf{0}_{2p} & \text{if } \mathcal{M}_{c+2}=\{1\}\\ \frac{c}{|\mathcal{M}_{c+2}|}\sum_{i\in[c]}\mathbf{1}_{\{E(C_i)\cdot E_{\text{not-false}}(A)=1\}}\cdot E(C_i) & \text{if } \mathcal{M}_{c+2}\subset[2,c+2]\end{cases}\\ &= m\sum_{i\in[c]}\mathbf{1}_{\{E(C_i)\cdot E_{\text{not-false}}(A)=1\}}\cdot E(C_i)\\ &= m\boldsymbol{z}\end{aligned}$$

for $m=\frac{c}{|\mathcal{M}_{c+2}|}>1$.

We now construct the weights for the ReGLU MLP layer. By Lemma C.1 we know that ReGLU MLP can simulate ReLU MLPs. Therefore, we only need to construct $\boldsymbol{W}_1^D,\boldsymbol{W}_2^D,\boldsymbol{b}_1^D,\boldsymbol{b}_2^D$ such that

$$\boldsymbol{W}_2^D\ \text{ReLU}(\boldsymbol{W}_1^D[m\boldsymbol{z};\boldsymbol{X}_{c+2}]+\boldsymbol{b}_1^D)+\boldsymbol{b}_2^D=\text{ReLU}(m\boldsymbol{z}-E_{\text{assigned}}(A))\ -\ \text{ReLU}(m\boldsymbol{z}-1).$$

Note that $\boldsymbol{X}_{c+2}=[E(A)\quad 0\quad 1]$, therefore $[m\boldsymbol{z};\boldsymbol{X}_{c+2}]\in\mathbb{R}^{4p+2}$ Also,

$$E_{\text{assigned}}(A)=\begin{bmatrix}\boldsymbol{I}_p & \boldsymbol{I}_p\\ \boldsymbol{I}_p & \boldsymbol{I}_p\end{bmatrix}E(A)$$

Therefore, define

$$\boldsymbol{W}_1^D=\begin{bmatrix}\boldsymbol{I}_{2p} & \mathbf{0}_{2p\times 2p} & -\begin{bmatrix}\boldsymbol{I}_p & \boldsymbol{I}_p\\ \boldsymbol{I}_p & \boldsymbol{I}_p\end{bmatrix} & \mathbf{0}_{2p\times 2}\\ \boldsymbol{I}_{2p} & \mathbf{0}_{2p\times 2p} & \mathbf{0}_{2p\times 2p} & \mathbf{0}_{2p\times 2}\end{bmatrix}$$

$$\boldsymbol{b}_1^D=\begin{bmatrix}\mathbf{0}_{2p}\\ -\mathbf{1}_{2p}\end{bmatrix}$$

$$\boldsymbol{W}_2^D=\begin{bmatrix}\boldsymbol{I}_{2p} & -\boldsymbol{I}_{2p}\end{bmatrix}$$

$$\boldsymbol{b}_2^D=\mathbf{0}_{2p}$$

It can be easily verified that this satisfies the desired equality. □

C.7. Theoretical Construction (Theorem 4.5)

Notations

- p denotes the number of variables
- t_i denotes the token at position i
- T_{vars} denotes the set of tokens that denote variables and their negations. i.e. '1', '2', ..., 'n', '-1', '-2', ..., '-n'
- b denotes boolean variables

Proof. We first describe the encoding format of the formulas and the solution trace format before going into the details of model construction.

Input Format. We consider 3-CNF-SAT formulas in the DIMACS representation, with an initial `[BOS]` token and an ending `[SEP]` token. Each variable x_i for $i \in [n]$ has 2 associated tokens: `i` and `-i` (e.g., `1` and `-1`), where the positive token indicates that the i-th variable appears in the clause while the negative token indicates that the negation of the i-th variable appears in the clause. Clauses are separated using the `0` token. For example, the formula

$$(\neg x_2 \vee \neg x_4 \vee \neg x_1) \wedge (x_3 \vee x_4 \vee \neg x_1) \wedge (\neg x_1 \vee \neg x_3 \vee \neg x_2) \\ \wedge (x_1 \vee \neg x_2 \vee \neg x_4) \wedge (\neg x_4 \vee x_2 \vee x_1) \wedge (x_1 \vee \neg x_2 \vee x_4)$$

would be represented as:

```
[BOS] -2 -4 -1 0 3 4 -1 0 -1 -3 -2 0 1 -2 -4 0 -4 2 1 0 1 -2 4 0 [SEP]
```

Solution Trace Format. The trace keeps track of the order of the assignments made and whether each assignment is a decision (assumption) or a unit propagation (deduction). Literals with a preceding `D` token are decision literals while other literals are from unit propagation. When the model encounters a conflict between the current assignment and the formula, it performs a backtrack operation denoted by `[BT]` and performs another attempt with the last decision literal negated. In particular, compared to Figure 1, we used `D` to abbreviate `Assume` and use `[BT]` to abbreviate `Backtrack`

As an example, the solution trace for the above SAT formula would be:
`[SEP] D 2 D 1 -4 3 [BT] D 2 -1 -4 [BT] -2 D 3 D 4 1 SAT` We use simplified versions of the tokens compared to Figure 1. In particular, we use `[BT]` as a shorthand for `BackTrack` and `D` for `Deduce`.

Implementation of Integer Arithmetic, Comparison, and Logical Operations In this paragraph, we show how basic arithmetic and comparison operations can be implemented as MLP layers. For simplicity, when describing the construction, we directly specify the arithmetic and comparison operations performed at each layer when applicable. In all descriptions below, we assume that we are given a vector of input parameters

$$\boldsymbol{X} = \begin{bmatrix} a_1 & b_1 \\ a_2 & b_2 \\ \vdots & \vdots \\ a_n & b_n \end{bmatrix} \in \mathbb{Z}^{n \times 2}$$

and describe the required MLP parameters to implement element-wise integer arithmetic between a_i and b_i at each position $i \in [n]$. For simplicity of expression, since the operations are element-wise, we assume wlog that the input is simply $[a; b] \in \mathbb{Z}^{1 \times 2}$.

- Addition $a + b$: Let
$$\boldsymbol{W}_1^+ = \begin{bmatrix} 1 & 1 \\ 0 & 0 \end{bmatrix} \quad \boldsymbol{b}_1^+ = \begin{bmatrix} 0 \\ 1 \end{bmatrix} \quad \boldsymbol{W}_2^+ = [1] \quad \boldsymbol{b}_2^+ = [0]$$
Then $\boldsymbol{W}_1^+[a; b] + \boldsymbol{b}_1^+ = [a + b; 1]$, $\mathrm{ReGLU}([a + b; 1]) = a + b$, $\boldsymbol{W}_2^+ \mathrm{ReGLU}([a + b; 1]) + \boldsymbol{b}_2^+ = a + b$
- Subtraction $a - b$: Implemented as $a + (-b)$:
$$\boldsymbol{W}_1^+ = \begin{bmatrix} 1 & -1 \\ 0 & 0 \end{bmatrix} \quad \boldsymbol{b}_1^+ = \begin{bmatrix} 0 \\ 1 \end{bmatrix} \quad \boldsymbol{W}_2^+ = [1] \quad \boldsymbol{b}_2^+ = [0]$$
- Multiplication $a \times b$: Apply Lemma C.3 with $\boldsymbol{x}_1 = \mathbf{a}$ and $\boldsymbol{x}_2 = \mathbf{b}$
- Comparison $\mathbf{1}_{a \le b}$: Let
$$\boldsymbol{W}_1^{\text{comp}} = \begin{bmatrix} 0 & 0 \\ 0 & 0 \\ -1 & 1 \\ -1 & 1 \end{bmatrix} \quad \boldsymbol{b}_1^{\text{comp}} = \begin{bmatrix} 1 \\ 1 \\ 0 \\ -1 \end{bmatrix} \quad \boldsymbol{W}_2^{\text{comp}} = \begin{bmatrix} 1 & -1 \end{bmatrix} \quad \boldsymbol{b}_2^{\text{comp}} = [0]$$

Then for input $[a, b]$, we have

$$\boldsymbol{W}_1^{\text{comp}}[a; b] + \boldsymbol{b}_1^{\text{comp}} = \begin{bmatrix} 1 \\ 1 \\ b-a \\ b-a-1 \end{bmatrix},$$

which is split into value part $\begin{bmatrix} 1 \\ 1 \end{bmatrix}$ and gate part $\begin{bmatrix} b-a \\ b-a-1 \end{bmatrix}$. Applying the ReGLU activation

$$\text{ReGLU}\left(\begin{bmatrix} 1 \\ 1 \\ b-a \\ b-a-1 \end{bmatrix}\right) = \begin{bmatrix} 1 \cdot \text{ReLU}(b-a) \\ 1 \cdot \text{ReLU}(b-a-1) \end{bmatrix} = \begin{bmatrix} \text{ReLU}(b-a) \\ \text{ReLU}(b-a-1) \end{bmatrix},$$

and finally

$$\boldsymbol{W}_2^{\text{comp}}\ \text{ReGLU}(\cdot) + \boldsymbol{b}_2^{\text{comp}} = \text{ReLU}(b-a) - \text{ReLU}(b-a-1) = 1_{a \le b}\,.$$

- Logical AND $a \wedge b$ where $a, b \in \{0, 1\}$: This is equal to $a \times b$
- Equivalence $a = b$: This takes 2 MLP layers to compute $(a \le b) \wedge (b \le a)$

Embedding Layer. Our token set consists of one token for each variable and its negation, the separator token `0`, and a special token `D` to denote where decisions are made. The positional encoding occupies a single dimension and contains the numerical value of the position of the token in the string. (i.e. there exists a dimension pos such that the position embedding of position i is $i \cdot \mathbf{e}_{pos}$)

Layer 1. The first layer prepares for finding the nearest separator token and D token. Let i denote the position index of tokens. Only MLP layers are used

1. Compute i_{sep} where $i_{\text{sep}} = i$ if the corresponding token $t_i \in \{\text{`0'}, \text{`[SEP]'}, \text{`[BT]'}\}$ and $i_{\text{sep}} = 0$ otherwise. This is computed as $i_{\text{sep}} = i \times (\boldsymbol{e}_{id(\texttt{0})} + \boldsymbol{e}_{id(\texttt{[SEP]})} + \boldsymbol{e}_{id(\texttt{[BT]})})$
2. Similarly, compute $i_{\texttt{D}}$ where $i_{\texttt{D}} = i$ if the corresponding token $t_i = \texttt{D}$ and $i_{\text{sep}} = 0$ otherwise. This is computed as $i_{\texttt{D}} = i \times \boldsymbol{e}_{id(\texttt{D})}$
3. Compute $(i-1)^2$, i^2 for index equality comparison

Layer 2. This layer uses 2 heads to perform the following tasks:

1. Copy the index and type of the last separator token and stores

$$\begin{aligned} p_i^{sep\prime} &= \max\{j : j \le i, t_j \in \{\text{`0'}, \text{`[SEP]'}, \text{`[BT]'}\}\} \\ b_{\texttt{0}} &= (t_j = \text{`0'}) \\ b_{\texttt{[SEP]}} &= (t_j = \text{`[SEP]'}) \\ b_{\texttt{[BT]}} &= (t_j = \text{`[BT]'}) \end{aligned}$$

   for $j = p_i^{sep\prime}$
2. (Backtrack) Compute the position of the nearest `D` token $p_i^{\texttt{D}} = \max\{j : j \le i, t_j = \text{`D'}\}$
3. Compute $(p_i^{sep\prime})^2$ for index operation

Task 1 can be achieved via the COPY operation from Lemma C.5 with $\boldsymbol{q}_i = 1$, $\boldsymbol{k}_i = i_{\text{sep}}$, $\boldsymbol{v}_j = (j, \mathbb{I}[t_j = \text{`0'}], \mathbb{I}[t_j = \text{`[SEP]'}], \mathbb{I}[t_j = \text{`[UP]'}], \mathbb{I}[t_j = \text{`[BackTrack]'}])$.

Task 2 is highly similar to task 1 and can be achieved using COPY with $\boldsymbol{q}_i = 1$, $\boldsymbol{k}_i = i_{\texttt{D}}$, $\boldsymbol{v}_j = (j)$

Layer 3 This layer uses 1 head to copy the several values from the previous token to the current token. Specifically, this layer computes:

1. The position of the *previous* separator token, not including the current position:
$$p_i^{sep} = \max\{j : j < i, t_j \in \{\text{`0'}, \text{`[SEP]'}, \text{`[UP]'}, \text{`[BackTrack]'}\}\}$$
2. Dermine if the previous token is D: $b_{decision} = (t_{i-1} = \text{`D'})$ i.e., whether the current token is a decision variable
3. (Induction) Compute the offset of the current token to the previous separator token $d_i^{sep} = i - p_i^{sep\prime}$
4. Compute $(p_i^{sep})^2$, for equality comparison at the next layer.

Task 1 and 2 is done by copying $p_i^{sep\prime}$ and $\mathbb{I}[t_i = \text{`D'}]$ from the previous token. Specifially, we use the COPY operation from Lemma C.5 with $\boldsymbol{q}_i = ((i-1)^2, i-1, 1)$ and $\boldsymbol{k}_j = (-1, 2j, -j^2)$ which determines $i - 1 = j$ via $-((i-1)-j)^2 = 0$ and $\boldsymbol{v}_j = (p_i^{sep\prime}, \mathbb{I}[t_i = \text{`D'}])$.

Layer 4. This layer uses 2 heads to perform the following tasks:

1. Compute
$$E(B_i) = \sum_{j > p_i^{sep}, t_j \in T_{vars}} \mathbf{e}_{id(t_j)} = \sum_{p_j^{sep} = p_i^{sep}, t_j \in T_{vars}} \mathbf{e}_{id(t_j)}$$
Recall that $E(B_i)$ is a binary encoding of the literals in a clause or partial assignment. Since each literal is represented in the DIMACS encoding as a single token, summing up the one-hot token embedding of each literal in a clause/partial assignment is equivalent to computing its encoding $E(B)$ value. The above operator finds the previous separator and sums up all the embedding dimensions corresponding to literal one-hot embeddings. Therefore, the above computation satisfied:
   - **For input DIMACS encoding:** Suppose that the separator 0 token following clause k is at position i in the DIMACS encoding tokens, then $E(B_i) = E(C_k)$, where $t_j = \text{`0'}$
   - **For the current position:** For the final position l, for which we're predicting the next token, $E(B_l) = E(A)$, where A is the current partial assignment.
2. (Induction) Compute the position of the second-to-last separator $p_i^{sep-} = \max\{j : j < p_i^{sep}, t_j \in \{\text{`0'}, \text{`[SEP]'}, \text{`[BackTrack]'}\}\} = p_{p_i^{sep\prime}}^{sep\prime}$ and the corresponding current position in the previous state $p_i^- = p_i^{sep-} + d_i^{sep}$. As a special case for the first state, we also add 4 to p_i^- if $b_{\texttt{[SEP]}}$ is true, i.e. $p_i^- = p_i^{sep-} + d_i^{sep} + 4 \cdot b_{\texttt{[SEP]}}$. The additional 4 is the number of variables per clause + 1 to ensure that we don't consider the last clause as an assignment.
3. (Backtrack) Compute the position of the nearest D token to the last separator token $p_i^{\text{D}-} = p_{p_i^{sep\prime}}^{\text{D}}$
4. Compute $b_{exceed} = (p_i^- > p_i^{\text{D}-} + 1)$, this denotes whether we're beyond the last decision of the previous state.
5. Compare $(p_i^{\text{D}-} \leq p_i^-)$ for $b_{\text{BT\_finished}}$ at the next layer.
6. Compare if $p_i^{\text{D}-} = p_i^-$ for the $b_{backtrack}$ operator.
7. Compute $b'_{copy} = (p_i^- < p_i^{sep\prime} - 1)$

Task 1 is achieved using a MEAN operation with $\boldsymbol{q}_i = ((p_i^{sep})^2, p_i^{sep}, 1)$, $\boldsymbol{k}_j = ((-1, 2p_j^{sep}, -(p_j^{sep})^2)$, $\boldsymbol{v}_j = \mathbf{e}_{id(t_j)}$ for $t_j \in T_{vars}$. This attention operations results in $\frac{\mathbf{r}_i}{i - p_i^{sep}}$ The MLP layer then uses Lemma C.3 to multiply the mean result by $i - p_i^{sep}$ to obtain the $\mathbf{r}_i$.

Task 2 is achieved using the COPY operation with $\boldsymbol{q}_i = ((p_i^{sep})^2, p_i^{sep}, 1)$, $\boldsymbol{k}_j = (-1, 2j, -j^2)$ and $\boldsymbol{v}_j = p_i^{sep\prime}$. The MLP layer then performs the addition operation the computes p_i^- by Lemma C.2

Similarly, Task 3 is achieved using the COPY operation with $\boldsymbol{q}_i = ((p_i^{sep})^2, p_i^{sep}, 1)$, $\boldsymbol{k}_j = (-1, 2j, -j^2)$ and $\boldsymbol{v}_j = p_i^{\text{D}}$.

Layer 5. The third layer uses 5 heads to perform the following tasks:

1. Compute $\mathbf{1}_{A \models F}$, $\mathbf{1}_{F \models \neg A}$, $E(D)$ where $D := \{l \in L \mid F \wedge A \models_1 l\}$ according to Lemma 4.7 with the $E(B_i)$ values from the previous layer.
2. Compute $b_{final} = b_{exceed} \wedge b_{decision}$
3. Compare $b_{no\_decision} = (p_i^{\texttt{D}} \leq p_i^{sep})$, which denotes whether the current state contains *no* decision variables
4. Compute $b_{\text{BT\_finished}} = (p_i^{\texttt{D}-} \leq p_i^{-}) \wedge b_{\texttt{[BackTrack]}}$
5. Compare p_i^- with $p_i^{\texttt{D}-} - 1$ by storing $p_i^- \leq p_i^{\texttt{D}-} - 1$ and $p_i^- \geq p_i^{\texttt{D}-} - 1$ (to check for equality at the next layer)
6. Compare $b_{\text{backtrack}} = (p_i^- = p_i^{\texttt{D}-} - 1)$

Layer 6 This layer does the remaining boolean operators required for the output. In particular,

- $b_{unsat} = b_{no\_decision} \wedge b_{cont}$
- $b_{\texttt{[BT]}} = b_{cont} \wedge \neg(t_i = \texttt{[BT]})$
- Compute a vector that is equal to $b_{backtrack} \cdot \mathbf{e}_{BT}$, which is equal to $\mathbf{e}_{BT}$ if $b_{backtrack}$ is True and $\mathbf{0}$ otherwise. This is to allow the operation at the output layer for backtracking

Note that $\wedge$ can be implemented as a single ReLU operation for tasks 1 and 2 that can be implemented with Lemma C.1, and task 3 is a multiplication operation implemented with Lemma C.3

Layer 7 This layer performs a single operation with the MLP layer: Compute $b_{copy} \cdot e_{copy}$, which gates whether e_{copy} should be predicted based on b_{copy}. This enables condition 5 at the output layer.

Output Projection The final layer is responsible for producing the output of the model based on the computed output of the pervious layers. We constructed prioritized conditional outputs, where the model outputs the token according to the first satisfied condition in the order below:

1. If b_{sat} output `SAT`
2. If $b_{cont} \wedge b_{no\_decision}$ output `UNSAT`
3. If $b_{cont} \wedge \neg(t_i = \texttt{[BackTrack]})$ output '`[BackTrack]`'
4. (BackTrack) If $b_{backtrack}$, output the negation of the token from position $p_i^{\texttt{D}-} + 1$
5. (Induction) If b_{copy}, copy token from position $p_i^- + 1$ as output (e_{copy})
6. output a unit propagation variable, if any.
7. output `D` if the current token is not `D`
8. output a unassigned variable

For the output layer, we use $l_{\texttt{[TOKEN]}}$ to denote the output logit of `[TOKEN]`. Since the final output of the model is the token with the highest logit, we can implement output priority by assigning outputs of higher priority rules with higher logits than lower priority rules. Specifically, we compute the output logits vector using the output layer linear transformation as:

$$2^7 \cdot b_{sat} \cdot \mathbf{e}_{\texttt{SAT}} + 2^6 \cdot b_{cont} \cdot \mathbf{e}_{\texttt{[BackTrack]}} + 2^5 \cdot b_{unsat} \cdot \mathbf{e}_{\texttt{UNSAT}}$$
$$+2^4 \cdot b_{backtrack} \cdot \mathbf{e}_{BT} + 2^3 \cdot b_{copy} \cdot \mathbf{e}_{copy} + 2^2 \cdot \mathbf{e}_{\texttt{UnitPropVar}} + 2^1 \cdot (1 - \mathbf{1}[t_i = \text{`}\texttt{D}']) \cdot \mathbf{e}_{\texttt{D}} + 2^0 \cdot T[(0,0),(0,0),(1,1)]\mathbf{r}_i$$

Composing the Transformer In the above construction, we demonstrate how each operation can be approximated by a Self-attention or MLP layer. We can set the embedding dimension to the sum of dimensions of all the intermediate values and allocate for every intermediate values a range of dimensions that's equal to the dimension of the variables. All dimensions are initialized to 0 in the positional encoding of the transformer except for the dimensions assigned to the positional index i. Similarly, only the dimensions assigned to the one-hot token representation are initialized in the token embeddings. At each layer, the self-attention heads and MLP layers extract the variable values from the residual stream and perform the operations assigned to them at each layer.

□

Proposition C.11. *There exists a transformer with 7 layers, 5 heads, $O(p)$ embedding dimension, and $O(p^2)$ weights that, on all inputs $\boldsymbol{s} \in \mathrm{DIMACS}(p, c)$, predicts the same token as the output as the above operations. Furthermore, let $l_{ctx} = 4c + p \cdot 2^p$ be the worst-case maximum context length required to complete SAT-solving, then all weights are within $\mathrm{poly}(l_{ctx})$ and can be represented within $O(p + \log c)$ bits.*

We only argue from a high level why this is true due to the complexity of the construction.

The only intermediate values whose dimensions are dependent on p are the vectors for one-hot encodings and storing binary encodings of clauses and assignments. They all have size $2p$. Therefore, the number of total allocated embedding sizes is also $O(p)$.

Furthermore, Appendix C.4 shows that all parameter values are polynomial with respect to the context length and the inverse of approximation errors. Note that we need only guarantee the final error is less than 1 to prevent affecting the output token. Furthermore, we can choose all parameter values so that they are multiples of 0.5. As such, all parameters are within $\mathrm{poly}(l_{ctx})$ and can be represented by $O(\log(l_{ctx})) = O(p + \log c)$

C.8. Correctness of Construction (Theorem 4.5)

Note: This section assumes prior knowledge in propositional logic and SAT solving, including an understanding of the DPLL algorithm. For a brief explanation of the notations in this section, please refer to ((Nieuwenhuis et al., 2005)). For more general knowledge, please refer to ((Biere et al., 2009)).

We prove that the above model autoregressive solves 3-$\mathrm{SAT}_{p,c}$ by showing that it uses the CoT to simulate the "Abstract DPLL Procedure".

C.8.1. Abstract DPLL

In this section, we provide a description of abstract DPLL. Since the focus of this paper is not to show the correctness of the DPLL algorithm but rather how our model's CoT is equivalent to it, we only present the main results from (Nieuwenhuis et al., 2005) and refer readers to the original work for proof of the theorems.

Let M be an ordered trace of variable assignments with information on whether each assignment is an *decision literal* (i.e. assumption) or an *unit propagation* (i.e., deduction).

For example, the ordered trace $3^d\ 1\ \overline{2}\ 4^d\ 5$ denotes the following sequence of operations:

Assume $x_3 = T \rightarrow$ Deduce $x_1 = T \rightarrow$ Deduce $x_2 = F \rightarrow$ Assume $x_4 = T \rightarrow$ Deduce $x_5 = T$.

Let F denote a SAT formula in CNF format (which includes 3-SAT), C denote a clause (e.g., $x_1 \vee \neg x_2 \vee x_3$), l denote a single literal (e.g., $\neg x_2$), and l^d denote a decision literal. Let $M \models F$ denote that the assignment in M satisfies the formula F.

Definition C.12 (State in the DPLL Transition System)**.** A *state* $S \in \mathbb{S}$ in the DPLL transition system is either:

- The special states $\mathrm{SAT}, \mathrm{UNSAT}$, indicating that the formula satisfiable or unsatisfiable
- A pair $M \parallel F$, where:
  - F is a finite set of clauses $C_1 \wedge C_2 \cdots \wedge C_c$ (a conjunctive normal form (CNF) formula), and
  - M is a sequence of annotated literals $l_1 \circ l_2 \cdots \circ l_i$ for some $i \in [n]$ representing variable assignments, where $\circ$ denotes concatenation. Annotations indicate whether a literal is a decision literal (denoted by l^{d}) or derived

through unit propagation.

We denote the empty sequence of literals by $\emptyset$, unit sequences by their only literal, and the concatenation of two sequences by simple juxtaposition. While M is a sequence, it can also be viewed as a set of variable assignments by ignoring annotations and order.

Definition C.13 (Adapted from Definition 1 of (Nieuwenhuis et al., 2005))**.** The Basic DPLL system consists of the following transition rules $\mathbb{S} \Longrightarrow \mathbb{S}$:

UnitPropagate :

$$M \parallel F \wedge (C \vee l) \quad \Longrightarrow \quad M \circ l \parallel F \wedge (C \vee l) \quad \textbf{if} \quad \begin{cases} M \models \neg C, \\ l \text{ is undefined in } M. \end{cases}$$

Decide :

$$M \parallel F \quad \Longrightarrow \quad M \circ l^{\mathrm{d}} \parallel F \quad \textbf{if} \quad \begin{cases} l \text{ or } \neg l \text{ occurs in a clause of } F, \\ l \text{ is undefined in } M. \end{cases}$$

Backjump :

$$M \circ l^{\mathrm{d}} \circ N \parallel F \quad \Longrightarrow \quad M \circ l' \parallel F \quad \textbf{if} \quad \begin{cases} \text{There is some clause } C \vee l' \text{ s.t.} \\ F \models C \vee l', \quad M \models \neg C, \\ l' \text{ is undefined in } M, \\ l' \text{ or } \neg l' \text{ occurs in a clause of } F. \end{cases}$$

Fail :

$$M \parallel F \wedge C \quad \Longrightarrow \quad \text{UNSAT} \quad \textbf{if} \quad \begin{cases} M \models \neg C, \\ M \text{ contains no decision literals.} \end{cases}$$

Success :

$$M \parallel F \quad \Longrightarrow \quad \text{SAT} \quad \textbf{if} \quad M \models F$$

We also use $S \Longrightarrow^* S'$ to denote that there exist $S_1, S_2, \ldots, S_i$ such that $S \Longrightarrow S_1 \Longrightarrow \cdots \Longrightarrow S_i \Longrightarrow S'$. Also $S \Longrightarrow^! S'$ denote that $S \Longrightarrow^* S'$ and S' is a final state (SAT or UNSAT).

Explanation of the Backjump *Operation:*

The Backjump operation allows the DPLL algorithm to backtrack to a previous decision and learn a new literal. In particular, $F \models C \vee l'$ means that, for some clause C, every assignment that satisfies F must either satisfy C (i.e., contain the negation of each literal in C) or contain l' as an assignment. However, if $M \models \neg C$, which means that M conflicts with C and thus contains the negation of each literal in C, then if we want some assignment containing M to still satisfy F, then the assignment must also include the literal l' as an assignment to ensure that it satisfies $C \vee l'$, a requirement for satisfying F.

In our construction, we only consider the narrower set of BackTrack operations that find the last decision and negate it:

Lemma C.14. *[Corrollary of Lemma 6 from (Nieuwenhuis et al., 2005)] Assume that* $\emptyset \parallel F \Longrightarrow^* M \circ l^{\mathrm{d}} \circ N \parallel F$*, the* BackTrack *operation:*

$$M \circ l^{\mathrm{d}} \circ N \parallel F \quad \Longrightarrow \quad M \circ \neg l \parallel F \quad \textbf{\textit{if}} \quad \begin{cases} \textit{There exists clause } C \textit{ in } F \textit{ such that} \\ M \circ l^{\mathrm{d}} \circ N \models \neg C \\ N \textit{contains no decision literals} \end{cases}$$

is always a valid Backjump *operation in Definition C.13.*

Definition C.15 (Run of the DPLL Algorithm)**.** A *run* of the DPLL algorithm on formula F is a sequence of states $S_0 \Longrightarrow S_1 \Longrightarrow \cdots \Longrightarrow S_T$ such that:

- S_0 is the initial state $\emptyset \parallel F$

- For each $i = 0, 1, \ldots, n-1$, the transition $S_i \Longrightarrow S_{i+1}$ is valid according to the transition rules of the DPLL system in Definition C.13 (e.g., UnitPropagate, Decide, Backjump, or Fail);
- S_n is a final state that is either SAT or UNSAT

Note that the above definition is simply the expansion of $\emptyset \parallel F \Longrightarrow^{!} S_T$.

The following theorem states that the DPLL procedure always decides the satisfiability of CNF formulas:

Lemma C.16. *[Theorem 5 and Theorem 9 Combined from (Nieuwenhuis et al., 2005)] The Basic DPLL system provides a decision procedure for the satisfiability of CNF formulas F. Specifically:*

1. $\emptyset \parallel F \Longrightarrow^{!} \text{UNSAT}$ *if and only if F is unsatisfiable.*
2. $\emptyset \parallel F \Longrightarrow^{!} \text{SAT}$ *if and only if F is satisfiable.*
3. *There exist no infinite sequences of the form $\emptyset \parallel F \Longrightarrow S_1 \Longrightarrow \cdots$*

C.8.2. Trace Equivalence and Inductive Proof

To prove that our Transformer indeed simulates abstract DPLL algorithm, we use an argument of refinement: we view our Transformer construction with CoT as a state transition system and show that that transitions of this system "refines" that of the abstract DPLL state transition system:

Definition C.17. A transition system is a tuple (S, T, s_0) where S is the set of states, $T \subseteq S \times S$ is the transition relation, and s_0 is the start state. If $(s_1, s_2) \in T$, we say that there is a transition from s_1 to s_2 and denote $s_1 \Rightarrow s_2$.

Definition C.18. A run r of transition system (S, T, s_0) is a (potentially infinite) sequence $(s_0, s_1, \ldots)$ such that:

- The sequence starts with s_0
- At each step $t \geq 0$, $(s_t, s_{t+1}) \in T$

The run r *halts* if it's a finite sequence such that $(s_0, s_1, \ldots, s_t^*)$ such that s_t^* does not have any next transitions, i.e., There's no state s such that $(s_t^*, s) \in T$

Definition C.19 (Refinement)**.** Given two transition systems $A = (S_A, T_A, s_{A0})$ and $B = (S_B, T_B, s_{B0})$. Transition system A *refines* B if there is a *refinement mapping* $R \subseteq S_A \times S_B$ such that:

1. R maps the initial state of A to the initial state of B:
$$(s_{A0}, s_{B0}) \in R.$$
2. For every $(s_A, s_B) \in R$, and every run r that contains s_A, let $r' = (s_A, \ldots)$ be the suffix of r starting from s_A. There exists $s'_A \in S_A$, $s'_B \in S_B$ such that $s'_A \in r'$ and $(s_B, s'_B) \in T_B$.

Here, $\Rightarrow_A^*$ denotes the reflexive transitive closure of $\Rightarrow_A$.

Proposition C.20. *Given two transition systems $A = (S_A, T_A, s_{A0})$ and $B = (S_B, T_B, s_{B0})$. If transition system A* refines *B, and every run of B halts and ends in state s_B^*, then every run of A contains on s_A^* such that $R(s_A^*) = s_B^*$.*

To proceed with this argument, we first need to define the refinement mapping between our model's CoT and the states of abstract DPLL. Consider the following model input and CoT trace:

```
[BOS] -2 -4 -1 0 3 4 -1 0 -1 -3 -2 0 1 -2 -4 0 -4 2 1 0 1 -2 4 0 [SEP] D 2 D 1 -4
3 [BT] D 2 D -1 -4
```

Recall that `[BT]` denotes backtracking and `D` denotes that the next token is a decision literal.

Note that the prompt input ends at `[SEP]` and the rest is the CoT produced by the model.

We want to convert this trace to a state $S = M \| F$ such that F is the CNF formula in the DIAMCS encoding in the prompt input and M is the "assignment trace" at the last attempt (i.e., after the last `[BT]` token.). As such, M correspond to the `D 2 D -1 -4` portion of the trace and thus $M = 2^d \, \bar{1}^d \, \bar{4}$ as described in Appendix C.8.1. We formalize this process as follows:

Definition C.21 (Translating CoT to Abstract DPLL State)**.** For any number of variables $p \in \mathbb{N}^+$, let $\mathcal{V}$ be the set of tokens:

$$\mathcal{V} = \{\texttt{-i},\ \texttt{i} \mid i \in [p]\} \cup \{\texttt{D},\ \texttt{[SEP]},\ \texttt{[BOS]},\ \texttt{[BT]},\ \texttt{0},\ \texttt{SAT},\ \texttt{UNSAT}\}.$$

Define a mapping $f_{\mathcal{S}} : \mathcal{V}^* \to \mathcal{S} \cup \{\text{error}\}$ that converts a sequence of tokens $R \in \mathcal{V}^*$ into an abstract DPLL state as follows:

1. **If** R ends with `SAT` or `UNSAT`, **then** set $M_{\mathcal{S}}(R)$ to SAT or UNSAT accordingly.
2. **Else if** R contains exactly one `[SEP]` token, split R at `[SEP]` into R_{DIMACS} and R_{Trace}.
3. Parse R_{DIMACS} as a DIMACS representation of CNF formula F, assuming it starts with `[BOS]` and ends with `0`. If parsing fails, set $M_{\mathcal{S}}(R) = \text{fail}$.
4. Find the last `[BT]` in R_{Trace}, and let R_{current} be the part of R_{Trace} after the last `[BT]`. If there's none, set R_{current} to R_{Trace}.
5. Initialize an empty sequence M to represent variable assignments and set a flag *isDecision* $\leftarrow$ `False`.
6. Process each token t in R_{current} sequentially:
   - **If** $t =$ `D`, set *isDecision* $\leftarrow$ `True`.
   - **Else if** l is a literal, append l to M, annotated as a decision literal if *isDecision* $=$ `True`, or as a unit propagation otherwise.
   - Reset *isDecision* $\leftarrow$ `False`.
   - **Else**, set $M_{\mathcal{S}}(R) = \text{error}$.
7. **Return** the state $M \parallel F$.

With the above mapping, we can specify the following properties of our Transformer construction based on logical relations between A and F:

Proposition C.22. *Given input sequence $s_{1:n} \in \mathcal{V}^*$ such that $f_{\mathcal{S}}(s_{1:n}) = M \parallel F$ for which F is a valid 3-SAT formula and M is a sequence of annotated literals. Let A be the partial assignment corresponding to M (i.e., removing annotation and order). Let $D := \{l \in L \mid F \wedge A \models_1 l\}$ be the set of literals that can be deduced through unit propagation. Let U be the set of literals corresponding to variables not assigned in A. Let s_{n+1} be the output of the Transformer model defined in Appendix C.7 when given $s_{1:n}$ as input, then s_{n+1} satisfy the following:*

$$\begin{aligned}
A \models F &\implies s_{n+1} = \texttt{SAT} \\
(M \text{ contains no decision literals}) \ \wedge\ (F \models \neg A) &\implies s_{n+1} = \texttt{UNSAT} \\
(M \text{ contains decision literals}) \ \wedge\ (F \models \neg A) &\implies s_{n+1} = \texttt{[BackTrack]} \\
(A \not\models F) \wedge (F \not\models \neg A) \wedge (D \neq \emptyset) &\implies s_{n+1} \in D \\
(A \not\models F) \wedge (F \not\models \neg A) \wedge (D = \emptyset) \wedge (s_n \neq \texttt{D}) &\implies s_{n+1} = \texttt{D} \\
(A \not\models F) \wedge (F \not\models \neg A) \wedge (D = \emptyset) \wedge (s_n = \texttt{D}) &\implies s_{n+1} \in U
\end{aligned}$$

We now present the inductive lemma:

Lemma C.23 (Inductive Lemma)**.** *For any $p, c \in \mathbb{N}^+$, for any input $F_{DIMACS} \in \text{DIMACS}(p, c)$ of length n, let F be the boolean formula in CNF form encoded in F_{DIMACS}. Let A be the model described in section C.7 with parameters p, c. Let $(\boldsymbol{s}_{1:n}, \boldsymbol{s}_{1:n+1}, \dots)$ be the trace of $\boldsymbol{s}$ when running the Greedy Decoding Algorithm 1 with model A and input prompt $\boldsymbol{s}_{1:n} = F_{DIMACS}$. For every $i \in \mathbb{N}^+$, if $f_{\mathcal{S}}(\boldsymbol{s}_{1:n+i}) = S$ and $S \notin \{\text{SAT}, \text{UNSAT}, \text{error}\}$, then there exist $j \in \mathbb{N}^+$ and $S' \in \mathbb{S}$ such that $S \implies S'$ and $f_{\mathcal{S}}(\boldsymbol{s}_{1:n+i+j}) = S'$.*

We now show trace equivalence between the model A and some instantiating of the abstract DPLL with a specific heuristic:

Definition C.24. For any heuristic $h : \mathbb{S} \to \mathcal{L}$ where $\mathcal{L}$ is the set of literals, let DPLL_h denote an instantiation of the abstract DPLL algorithm that selects $h(S)$ as the decision literal when performing Decide and only performs the BackTrack operation for Backjump. $h(S)$ is a valid heuristic if DPLL_h always abides by the Decide transition.

Lemma C.25. *(Trace Simulation) There exists a valid heuristic $h : \mathbb{S} \to \mathcal{L}$ for which the Transformer model A is trace equivalent to DPLL_h on all inputs in $\text{DIMACS}(p, c)$*

D. PARAT and Compiled Theoretical Construction

D.1. Supported Features and Operations

Our tool is designed to provide an intuitive syntax resembling standard numerical array manipulation, akin to NumPy, while supporting a diverse and extensible set of abstract operations. PARAT is capable of implementing

- **NumPy-like Array Syntax** for indexing, arithmetic, and comparison.
- **Multi-Level Abstraction** to enable low-level customization.
- **Multi-stage Evaluation Mechanisms** to facilitate debugging and error localization
- **High Extensibility** through structured class inheritance, promoting the addition of new features and operations.

Each intermediate "variable" is an instance of the SOp base class (name adapted from (Lindner et al., 2023)), and each instance sop of SOp is assigned a dimension $d_{\texttt{sop}} \in \mathbb{N}^+$ and can be viewed as an abstract representation of an $\mathbb{R}^{n \times d_{\texttt{sop}}}$ array where n is the number of tokens in the input to the Transformer model. A PARAT "program" is basically a sequence of array operations over SOps.

Throughout this section, we refer to the indices along the first dimension of an SOp as "position" and refer to indices along the second dimension as "dimension".

The "inputs" to a program are arbitrary positional encoding and token embedding variables, represented by the base class names PosEncSOp and TokEmbSOp respectively. For example, the OneHotTokEmb class represents the one-hot embedding of tokens and Indices represents the numerical value of the index of each position.

The rest of the program performs various operations that compute new SOps based on existing ones. We provide implementations of basic building block operations including (but not limited to) the following:

- `Mean(q, k, v)` Represents the "Averaging Hard Attention" operation. At each position i, this operation identifies all target positions j with the maximal value of $q_i^\top k_j$ for $j \leq i$ and computes the average of the corresponding v_j values.
- `sop[idx, :]` Performs indexing using a one-dimensional index array `idx`, producing an SOp `out` such that $\texttt{out}[i, j] = \texttt{sop}[\texttt{idx}[i], j]$ for $i \in [n]$ and $j \in [d_{\texttt{sop}}]$. This mirrors NumPy's array indexing semantics.
- `sop[:, start:end]` Extracts a slice of dimensions from sop, where $\texttt{start}, \texttt{end} \in [d_{\texttt{sop}}]$, resulting in a new SOp of dimension $\texttt{end} - \texttt{start}$. This operation is analogous to NumPy slicing.
- Element-wise operations such as `sop1 + sop2`, `sop1 - sop2`, `sop1 * sop2`, logical operations (`&` for AND, `|` for OR), and comparison operations ($\geq, \leq, >, <$), following standard broadcasting rules.

As an illustrative example, the following function returns a one-dimensional SOp representing the position index of the closest token within a set of target tokens:

```
def nearest_token_id(tok_emb: OneHotTokEmb, vocab: List[str],
                     targets: List[str], indices: Indices=indices):
    # Get the token ids of the target tokens
    target_tok_ids = [vocab.index(target) for target in targets]
    # Get whether the current token is one of the target tokens
    # by summing the one-hot embedding
    target_token_embs = Concat([tok_emb[:, target_tok_id]
                                for target_tok_id in target_tok_ids])
    in_targets = target_token_embs.sum(axis=1)
    # Filter the indices to only include the target tokens
    filtered_index = indices * in_targets
    return filtered_index.max()
```

We present our full code implementing our construction for Theorem 4.5 using PARAT in Appendix D.4.

D.2. Comparison with Tracr (Lindner et al., 2023)

While Tracr also compiles RASP programs into Transformer weights, the RASP language is designed to provide a concise description of the class of functions that Transformers can easily learn. As such, RASP has minimal syntax and is designed to represent relatively simple sequence operations such as counting, sorting, etc. In contrast, our tool is designed to help construct theoretical constructions that implement relatively more complex algorithms.

In our preliminary attempt to implement our SAT solver model with Tracr, we identified several implementation inconveniences and limitations of Tracr when scaling to more complex algorithms, which motivated the development of our tool. In particular:

- Every "variable" (termed `sop` in (Lindner et al., 2023)) in Tracr must be either a one-hot categorical encoding or a single numerical value. This constraint makes representing more complex vector structures highly inconvenient. Furthermore, each `select` operation (i.e., self-attention) accepts only a single `sop` as the query and key vectors, whereas our theoretical construction often requires incorporating multiple variables as queries and keys.

  In contrast, each variable in `PARAT` represents a 2-D array, which facilitates the implementation of vector-based operations such as performing logical deductions as described in Lemma 4.7

- In terms of parameter complexity, Tracr represents position indices and many other discrete `sop`s with a one-hot encoding, allocating a residual stream dimension for each possible value of the `sop`. In particular, compiling models with a context length of n requires $O(n)$ additional embedding dimensions for each SOp that represents a position index. For each binary operation between one-hot encoded `sop`s (such as position indices), Tracr creates an MLP layer that first creates a lookup table of all possible value combinations of the input `sop`s. This results in an MLP layer of $O(n^3)$ parameters.

  In contrast, our tool directly represents numerical values rather than working with token representations. For example, positional encodings only take up 1 dimension of the residual stream, which drastically reduces the number of parameters for longer context lengths.

We would like to emphasize that our goal is not to replace Tracr or RASP, which have unparalleled simplicity and interpretability in describing well-studied sequence operations. The goal of our tool is to assist with creating implementations of theoretical constructions to help verify its behaviors and investigate internal properties.

D.3. The Compilation Process

`PARAT` takes in an `out` variable that contains the computational graph of the algorithm and outputs a PyTorch ((Paszke et al., 2017)) model. The compilation process follows stages similar to those of Tracr:

1. **Computational Graph Construction**: When a user writes `sop` operations, each operation automatically creates a dependency tree of all operations required for computing the resulting `sop` value.

2. **Reduction to Base Operations:** Each `sop` operation is reduced to one of 5 base classes: `SelfAttention` for operation that requires information from other token positions, `GLUMLP` for non-linear local operations, `Linear` for linear local operations, `PosEncSOp` for positional encodings, or `TokEmbSOp` for token embeddings. Sequential `Linear` operations are reduced to a single operation through matrix multiplication and dependency merging.

3. **Allocation of Layers and Residual Stream:** The computational graph is topologically sorted such that each `sop` appears later than its dependencies. This sorting is then used to assign `SelfAttention` and `GLUMLP` `sop`s to Transformer layer numbers that comply with dependency constraints. Furthermore, each non-`Linear` `sop` is also allocated a portion of the residual stream equal to their $d_{\texttt{sop}}$ size.

4. **Model Construction and Weight Assignment:** A PyTorch model is initialized based on the number of required layers, hidden size, and embedding size inferred from the previous steps. The computed weights for each `sop` are assigned to different model components based on their types. Notably, each `SelfAttention` `sop` corresponds to an attention head, and each `GLUMLP` sops corresponds to part of a MLP layer with `ReGLU` activation.

Soft vs Hard Attention The reduction of `Mean` to `SelfAttention` induces inevitable numerical errors due to `Mean` representing averaging a strict subset of previous positions while `SelfAttention` computes a weighted average over all previous positions via softmax. This error also affects other operations based on `Mean` such as position indexing. We control this error via an "exactness" parameter β that scales the attention logits, and Lemma C.6 shows that the error decreases exponentially w.r.t. β.

Multi-Stage Evaluation To facilitate debugging, `PARAT` allows 3 types of evaluations for every `sop` at different stages of compilation.

- `sop.abstract_eval(tokens)` evaluates `sop` on a sequence of input tokens without any numerical errors. This can be used to validate the correctness of the algorithm implementation as `sop` operations.
- `sop.concrete_eval(tokens)` evaluates `sop` on an input sequence after reducing to the base classes at step 2 of the compilation process. This helps localize errors stemming from incorrect reduction of high-level operations to base classes.
- **Model evaluation** This corresponds to evaluating the Pytorch model after the full compilation process.

D.4. Code for Theoretical Construction

The following code is used to construct the Transformer specification passed as input to `PARAT`. To facilitate easier implementation, we interleave PARAT statements with Python and Numpy operations when appropriate. PARAT takes the return variable `out` as input and produces the theoretical construction discussed in Section 5.1

```

def nearest_token(tok_emb: OneHotTokEmb, vocab: List[str],
                  targets: List[str], v: SOp | List[SOp],
                  indices: PosEncSOp = indices):
    if not isinstance(v, list):
        v = [v]

    target_tok_ids = [vocab.index(target) for target in targets]
    target_tokens = Concat([tok_emb[:, target_tok_id]
        for target_tok_id in target_tok_ids])
    in_targets = Linear(target_tokens, np.ones((1, len(targets))))
    filtered_index = (indices * in_targets)

    new_v = []
    for v_i in v:
        if isinstance(v_i, SOp):
            new_v.append(v_i)
        elif v_i == 'target' or v_i == 'targets':
            new_v.append(target_tokens)
        else:
            raise ValueError('Unsupported value type')

    return Mean(ones, filtered_index, new_v, bos_weight=1)


def t(encodings: SOp, num_vars,
      true_vec=(1, 0),
      false_vec=(0, 1),
      none_vec=(0, 0),
      ones: Ones = ones):
    mat = np.zeros((2 * num_vars, 2 * num_vars))
    true_vec_off = (true_vec[0] - none_vec[0], true_vec[1] - none_vec[1])
    false_vec_off = (false_vec[0] - none_vec[0], false_vec[1] - none_vec[1])
    for i in range(num_vars):
        true_id = i
        false_id = num_vars + i
        mat[true_id, true_id] = true_vec_off[0]
```

```
        mat[true_id, false_id] = false_vec_off[0]
        mat[false_id, true_id] = true_vec_off[1]
        mat[false_id, false_id] = false_vec_off[1]

    bias = np.zeros(2 * num_vars)
    bias[:num_vars] += none_vec[0]
    bias[num_vars:] = none_vec[1]

    return Linear([encodings, ones],
        np.hstack([mat.T, bias.reshape((-1, 1))]))


def dpll(num_vars, num_clauses, context_len,
         mean_exactness=20, nonsep_penalty=20,
         return_logs=False) -> Tuple[
    SOp, List, Dict[str, SOp]]:
    vocab: List = ([str(i) for i in range(1, num_vars + 1)]
                   + [str(-i) for i in range(1, num_vars + 1)]
                   + ['0', '[SEP]', '[BT]', '[BOS]', 'D', 'SAT', 'UNSAT'])
    idx: Dict[str, int] = {token: idx for idx, token in enumerate(vocab)}
    sop_logs: Dict[str, SOp] = {}
    sops.config["mean_exactness"] = mean_exactness
    # Initialize Base SOps
    tok_emb = OneHotTokEmb(idx).named("tok_emb")

    nearest_sep = nearest_token(tok_emb=tok_emb,
                                vocab=vocab,
                                targets=['0', '[SEP]', '[BT]'],
                                v=[indices, 'target']).named(
        "nearest_sep")

    # The nearest (including self) separator token and whether
    # the previous separator token is '0', '[SEP]', '[UP]', '[BT]'
    p_i_sep_p, b_0, b_SEP, b_BackTrack = (
        nearest_sep[:, 0].named("p_i_sep_p"),
        nearest_sep[:, 1].named("b_0"),
        nearest_sep[:, 2].named("b_SEP"),
        nearest_sep[:, 3].named("b_BackTrack"))

    # The nearest 'D' token, which denotes the next token is a decision
    p_i_D = nearest_token(tok_emb=tok_emb, vocab=vocab, targets=['D'],
                          v=indices).named("p_i_D")

    prev_pos = Id([p_i_sep_p, tok_emb[:, idx['D']]])[indices - 1]
    # p_i_sep: The previous (excluding self) separator token
    p_i_sep = (prev_pos[:, 0] - is_bos).named("p_i_sep")

    # b_decision: whether the current position is a decision literal
    b_decision = prev_pos[:, 1].named("b_decision")

    # The distance to the nearest separator,
    # i.e., the length of the current state
    d_i_sep = (indices - p_i_sep_p).named("d_i_sep")

    # Attention operation for representing the current
    # clause/assignment as a bitvector of dimension 2d
    p_i_sep_2 = (p_i_sep * p_i_sep).named("p_i_sep_2")
    e_vars = tok_emb[:, : 2 * num_vars].named("e_vars")
    r_i_pre = Mean(q_sops=[p_i_sep_2, p_i_sep, ones],
                   k_sops=[-ones, 2 * p_i_sep, -p_i_sep_2],
                   v_sops=e_vars).named("r_i_pre")
    r_i = (r_i_pre * (indices - p_i_sep)).named("r_i")

    # The position of the previous (excluding self) separator token
```

```
    p_i_sep_min = p_i_sep[p_i_sep_p].named("p_i_sep_min")

    # The same position in the previous state.
    # This is used for copying from the previous state
    p_i_min = (p_i_sep_min + d_i_sep + num_vars * b_SEP).named("p_i_min")

    # The position of the last decision in the previous state
    p_i_D_min = p_i_D[p_i_sep_p].named("p_i_D_min")

    # Is the next token the literal resulting from backtracking?
    b_D_min = (p_i_D_min == p_i_min + 1).named("b_D_min")

    # Check if the current assignment satisfies the formula
    # (See Theorem Proof for justification)
    sat_q = [r_i, ones]
    sat_k = [-r_i, (-nonsep_penalty) * (1 - tok_emb[:, idx['0']])]
    sat_v = is_bos
    b_sat = (Mean(sat_q, sat_k, sat_v,
        bos_weight=nonsep_penalty - 0.5) > 0).named("b_sat")

    # Check if the current assignment contracdicts the formula
    # (See Theorem Proof for justification)
    unsat_q = [t(r_i, num_vars, true_vec=(1, 0),
        false_vec=(0, 1), none_vec=(1, 1)), ones]
    unsat_k = sat_k
    unsat_v = 1 - is_bos
    b_cont = (Mean(unsat_q, unsat_k, unsat_v,
        bos_weight=nonsep_penalty - 0.5) > 0).named("b_cont")
    b_copy_p = (p_i_min < (p_i_sep_p - 1)).named("b_copy_p")



    # Unit Propagation
    up_q = unsat_q
    up_k = unsat_k
    up_v = num_clauses * r_i
    o_up = Mean(up_q, up_k, up_v, bos_weight=nonsep_penalty - 1.5)


    e_up = (
            GLUMLP(act_sops=(o_up - t(r_i, num_vars,
                                      true_vec=(1, 1),
                                      false_vec=(1, 1),
                                      none_vec=(0, 0))))
            - GLUMLP(act_sops=(o_up - 1))
    ).named("e_up_new")


    # Heuristic for decision literal selection:
    # Find the most common literal in remaining clauses
    heuristic_q = [t(r_i, num_vars, true_vec=(-10, 1),
                     false_vec=(1, -10), none_vec=(0, 0)), ones]
    heuristic_k = [r_i, (-nonsep_penalty) * (1 - tok_emb[:, idx['0']])]
    heuristic_v = r_i
    heuristic_o = SelfAttention(heuristic_q, heuristic_k, heuristic_v)

    # Whether the current assignment contains no decision literal
    b_no_decision = (p_i_D <= p_i_sep).named("b_no_decision")

    # Whether Backtracking is finished
    b_BT_finish = ((p_i_D_min <= p_i_min) & b_BackTrack)

    # The negation of the last decision literal in the previous state
    e_BT = t(e_vars[p_i_D_min + 1], num_vars=num_vars,
```

```
            true_vec=(0, 1), false_vec=(1, 0), none_vec=(0, 0))

    # The next index in the previous state for copying
    p_i_min_index = (p_i_min + 1).named("p_i_min_index")

    # The next token in the previous state for copying
    e_copy = tok_emb[p_i_min_index].named("e_copy")

    # Whether we've decided that the formula is UNSAT
    b_unsat = (b_no_decision & b_cont).named("b_unsat")

    # Whether we're negativing the last decision literal for backtracking
    b_backtrack = (b_D_min & b_BackTrack).named("b_backtrack")

    # Whether we're copying tokens from the previous state
    b_copy = (b_copy_p & (1 - b_BT_finish)).named("b_copy")

    b_BT_token = (b_cont & (1 - tok_emb[:, idx['[BT]']]))
    b_not_D = (1 - tok_emb[:, idx['D']]).named("b_not_D")
    e_unassigned = t(r_i, num_vars, true_vec=(0, 0),
        false_vec=(0, 0), none_vec=(1, 1)).named("e_unassigned")

    out = CPOutput(len(vocab),
        [(b_sat, idx['SAT'], 16),
        (b_unsat, idx['UNSAT'], 15),
        (b_BT_token, idx['[BT]'], 14),
        (b_backtrack, Pad(e_BT, len(vocab), idx['1']), 12),
        (b_copy, e_copy, 6),
        (None, Pad(e_up, len(vocab), idx['1']), 4),
        (b_not_D, idx['D'], 3),
        (None, Pad(e_unassigned + heuristic_o,
                   out_dim=len(vocab), start_dim=idx['1']), 1)])

        return out
```