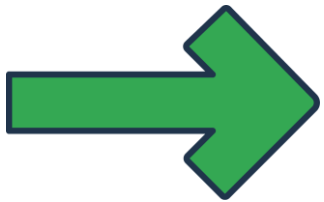
Google Developer Experts

LLM Constrained Decoding

Xihan Li, Ph.D.
Google Developer Expert in AI
Incoming Assistant Professor, Fudan University
https://snowkylin.github.io



Motivation: Strictly Structured Output for LLM

In industry scenarios, we often prompt LLMs to generate contents in structured format (JSON, SQL, Programming code, etc.)
But LLMs can violate these formats!

Examples

Text-to-SQL — Generating Database Queries

User Prompt	Expected Output	What LLM May Actually Output
Which users are older than 20?	`SELECT name FROM users WHERE age > 20;`	`SELECT name FROM user WEHRE age > 20;` table not exists; keyword misspelled

Tool / API Calls — Generating Function Arguments

Agent Task	Expected Output	What LLM May Actually Output
Call a weather API. The schema requires `city` as a string and `unit` as one of "celsius" or "fahrenheit"	`{"tool": "get_weather", "args": { "city": "London", "unit": "celsius" }}`	`{"tool": "get_weather", "args": { "city": London, "unit": "centigrade" }}` Missing quotes; not in enum

Examples

Code Completion — Generating Python Code

User Prompt	Expected Output	What LLM May Actually Output

Generate a python function that computes Fibonacci number.	`def fib(n):`
	`  if n <= 1:`
	`  return n`
	`  return fib(n-1) + fib(n-2)`
		`def fib(n):`
		`  if n <= 1:`
		`  return n`
		`  return fib(n-1) + fib(n-2))`

Indentation error

extra parenthesis

Information Extraction — Extracting Entities from Unstructured Text

User Prompt	Expected Output	What LLM May Actually Output

Extract structured information from "Alice joined the University of Bristol in 2024"	`{"name": "Alice",`
	`  "year": 2024,`
	`  "org": "University of Bristol"}`
		`Sure, let me analyze this sentence. The person mentioned is Alice, the year is 2024...`

Structured Output in Gemini API

```python
import google.generativeai as genai
import typing

# 1. Define your strict schema using Python's TypedDict
class Recipe(typing.TypedDict):
    recipe_name: str
    ingredients: list[str]
    prep_time_minutes: int

# 2. Initialize the model
model = genai.GenerativeModel('gemini-3-flash-preview')

# 3. Send the request, enabling the native schema feature
result = model.generate_content(
    "Give me a recipe for chocolate chip cookies.",
    generation_config=genai.GenerationConfig(
        response_mime_type="application/json", # Enable JSON mode
        response_schema=Recipe,                # Enforce the Recipe schema
    ),
)

print(result.text)
```

```json
{
  "recipe_name": "Classic Chocolate Chip Cookies",
  "prep_time_minutes": 15,
  "ingredients": [
    "2 1/4 cups all-purpose flour",
    "1 teaspoon baking soda",
    "1 teaspoon salt",
    ……
  ],
}
```

100% Guaranteed to be valid!
But how is this possible?

https://ai.google.dev/gemini-api/docs/structured-output

Recap: The LLM Generation Process

LLMs generate text token by token.

```
tokens ← [prompt_tokens]
while True:
    logits ← Model.forward(tokens)        // shape: [|V|], V is the vocabulary
    probs  ← softmax(logits)              // normalize to probability distribution
    next_token ← sample(probs)            // sample according to probabilities
    if next_token == EOS:
        break
    tokens.append(next_token)
return tokens
```

Position	1	2	3	4	5	6	7	8	9	10	11
Token	{	"	name	"	:	"	x	1	"	}	<EOS>

Constrained Decoding

Constrained decoding inserts an intervention step before softmax

```
tokens ← [prompt_tokens]
state ← grammar_automaton.initial_state     // initial state of the grammar automaton
while True:
    logits ← Model.forward(tokens)
    mask ← compute_mask(state, vocabulary)  // compute valid token mask from automaton
    logits ← logits + mask                  // apply mask ★
    probs  ← softmax(logits)
    next_token ← sample(probs)
    if next_token == EOS:
        break
    state ← grammar_automaton.advance(state, next_token)
    tokens.append(next_token)
return tokens
```

Mask: 0 = valid, −∞ = invalid

Vocabulary	{	}	:	"	name	x	0	1
Mask	0	−∞	−∞	0	−∞	−∞	0	0

The Concept of Grammar

Intuitively, “grammar” is something that tell us which sentences are correct and which are not

E.g., in English grammar

I eat food ✅ I food eat ❌

Similarly, we have “grammars” in computer science

Grammar	Valid example ✅	Invalid example ❌
Variable name	x0, y1	1a (cannot start with a digit)
Email	xihan.li@cs.ucl.ac.uk	xihan.li.at.cs.ucl.ac.uk (must be separated with an @)
JSON	{“a”: 1}	{“a”: b} (strings must be quoted)

The Concept of Grammar

Given a grammar, we can precisely determine whether a given string is valid or invalid under the grammar.

E.g., `Variable("x1") = True, Variable("1x") = False`

Level	Grammar Type	Equivalent Automaton	Typical Examples
Type 3	**Regular Grammar**	**Finite State Automaton (DFA/NFA)**	**Variable names, date formats, regex**
Type 2	Context-Free Grammar	Pushdown Automaton (PDA)	JSON, programming language syntax
Type 1	Context-Sensitive Grammar	Linear Bounded Automaton	Some natural language features
Type 0	Unrestricted Grammar	Turing Machine	All computable languages

Regular Grammar

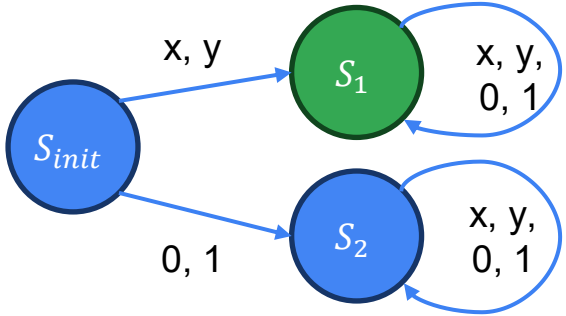
Regular Grammar can be expressed as a Determinist Finite Automaton (DFA)

DFA is a machine that process a sequence of characters (string):

- Starting from an initial state, it reads characters one by one.
- Each time it reads a character, it deterministically jumps to the next state based on the current state and the character read. `next_state = Transition(state, char)`
- When the last character is processed, check the state that the machine is on.
- Only return True if the state is in a pre-defined set F indicating that the string is valid.

Example: `Variable(string), vocabulary = {'0', '1', 'x', 'y'},` $F = \{S_1\}$

Rule:



Variable("1x"): $S_{init} \xrightarrow{1} S_2 \xrightarrow{x} S_2 \notin F$ = False

Variable("x1"): $S_{init} \xrightarrow{x} S_1 \xrightarrow{1} S_1 \in F$ = True

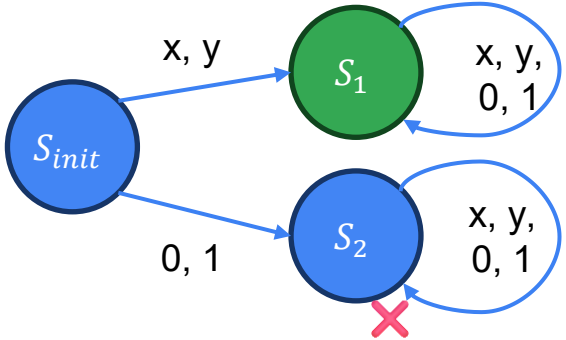
Pre-compute Live and Dead States

Live state: from this state, there exists at least one path that can reach some accept state in F.

Dead state: from this state, it is impossible to reach any accept state.

Example: `Variable(string), vocabulary = {'0', '1', 'x', 'y'},` $F = \{S_1\}$

Rule:



Live state: S_{init}, S_1

Dead state: S_2

DFA-Based Constrained Decoding

Assume every token in the LLM's vocabulary is exactly one character
E.g., `LLM.vocabulary = {'a', 'b', …, 'z', '0', '1', …}`

```
function constrained_decode(LLM, DFA):
    state ← DFA.init_state
    tokens ← []
    while True:
        logits ← LLM.forward(tokens)           // get score for each token
        for each token t in LLM.vocabulary:
            if DFA.transition(state, t) ∈ DFA.dead_states:  // target is dead state → mask
                logits[t] ← -∞
        next ← sample(softmax(logits))
        state ← DFA.transition(state, next)    // DFA move to the next state
        tokens.append(next)
        if state ∈ DFA.F and LLM wants to stop:
            break                              // valid termination
    return tokens
```

DFA-Based Constrained Decoding

In real LLMs, the vocabulary contains multi-character subword tokens
E.g., `LLM.vocabulary = {'apple', 'name', '3D', …}`

```
for each token t in LLM.vocabulary:
    if DFA.transition(state, t) ∈ DFA.dead_states:
    if is_token_valid(DFA, state, t):
        logits[t] ← -∞

function is_token_valid(DFA, state, token):
    for each character c in token:
        if DFA.transition(state, c) ∈ DFA.dead_states:
            return False
        state ← DFA.transition(state, c)
    return True
```

In real engineering, we can precompute a 2-dimensional matrix `valid[state, token] = is_token_valid(DFA, state, token, dead_states)`. See *Efficient Guided Generation for Large Language Models* (2023) for details

The Concept of Grammar

Given a grammar, we can precisely determine whether a given string is valid or invalid under the grammar.

E.g., `Variable("x1") = True, Variable("1x") = False`

Level	Grammar Type	Equivalent Automaton	Typical Examples
Type 3	Regular Grammar	Finite State Automaton (DFA/NFA)	Variable names, date formats, regex
Type 2	**Context-Free Grammar**	**Pushdown Automaton (PDA)**	**JSON, programming language syntax**
Type 1	Context-Sensitive Grammar	Linear Bounded Automaton	Some natural language features
Type 0	Unrestricted Grammar	Turing Machine	All computable languages

Context-Free Grammar

Regular Grammar has a fundamental limitation:

It cannot match arbitrary-depth nested structures

E.g., `{"a": {"b": {"c": …}}}` in JSON

And this is where Context-Free Grammar shines.

Here we focus on a minimal JSON Grammar example:

```
Json  → { Pair }      // JSON object = left brace + key-value pair + right brace
Pair  → Key : Val     // key-value pair = key + colon + value
Key   → "a" | "b"     // key has only two options
Val → 0 | 1 | Json    // value can be a number, or a nested JSON object ← recursion!
```

Non-terminal (in black): `Json, Pair, Key, Val`; Terminal (in red): `{, }, “a”, “b”, 0, 1`

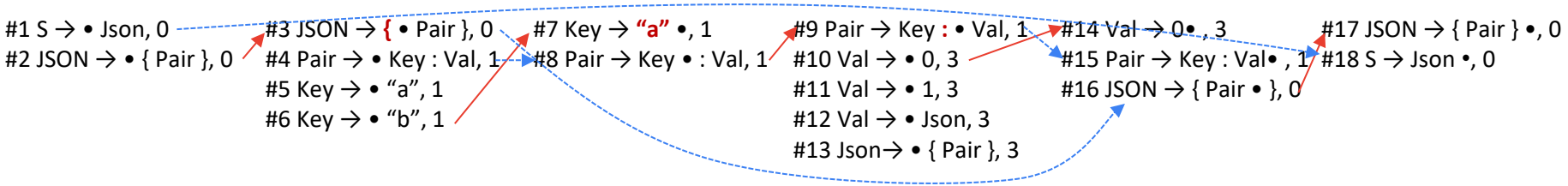
Similar to Regular Grammar and DFA, we first need to know how to validate a string programmatically.
Given a string (e.g., `{"a": 0}`), how do we know whether it is a valid minimal JSON?

The Earley Parser

```
Json → { Pair }
Pair → Key : Val
Key  → "a" | "b"
Val → 0 | 1 | Json
```

Let's check whether `{"a": 0}` is a valid minimal JSON, in a programmatic way
Earley Parser: Prediction, Scanning, Completion

0	1	2	3	4
{	"a"	:	0	}



Valid characters for position 0:
#2 {

Valid characters for position 1:
#5 "a"
#6 "b"

Valid characters for position 2:
#8 :

Valid characters for position 3:
#10 0
#11 1
#12 {

Valid characters for position 4:
#16 }

The input has been successfully parsed

Earley-Based Constrained Decoding

Assume every token in the LLM's vocabulary is exactly one character

```
function constrained_decode_cfg(LLM, grammar):
    earley_state ← initial_earley_set(grammar)
    tokens ← []
    while True:
        logits ← LLM.forward(tokens)
        valid_terminals ← extract_next_terminals(earley_state)
        for each token t in LLM.vocabulary:
            if t ∉ valid_terminals:
                logits[t] ← -∞
        next ← sample(softmax(logits))
        earley_state ← earley_advance(earley_state, next)
        tokens.append(next)
        if earley_state contains (S' → J •, 0):
            break
    return tokens
```

Earley-Based Constrained Decoding

In real LLMs, the vocabulary contains multi-character subword tokens

```
for each token t in LLM.vocabulary:
    if t ∉ valid_terminals:
    if is_token_valid_cfg(earley_state, t):
        logits[t] ← -∞

function is_token_valid_cfg(earley_state, token):
    current_state ← earley_state
    for each character c in token:
        valid ← extract_next_terminals(current_state)
        if c ∉ valid:
            return False
        current_state ← earley_advance(current_state, c)
    return True
```

Constrained Decoding Beyond JSON

For constrained decoding, you need

- A vocabulary V
- A predictive model $P(s_t|s_1, \dots, s_{t-1})$
- An `is_token_valid` function $F(s_1, \dots, s_t)$

Constrained decoding can do more than generating valid Json or SQL!

Problem Setting

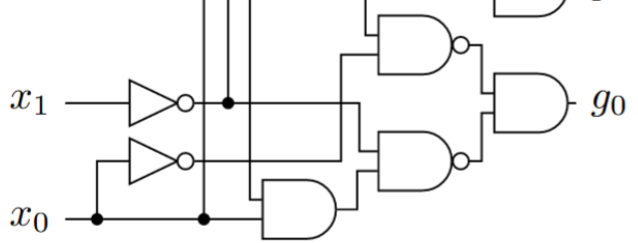
Implementing Boolean functions with logic circuits

(A fundamental problem in digital design!)

x_3	x_2	x_1	x_0	y_1	y_0	x_3	x_2	x_1	x_0	y_1	y_0
0	0	0	0	0	1	1	0	0	0	0	1
0	0	0	1	0	0	1	0	0	1	0	0
0	0	1	0	0	0	1	0	1	0	0	0
0	0	1	1	1	1	1	0	1	1	1	1
0	1	0	0	0	1	1	1	0	0	0	0
0	1	0	1	0	0	1	1	0	1	1	1
0	1	1	0	0	0	1	1	1	0	0	0
0	1	1	1	1	1	1	1	1	1	1	1

A Boolean function $(y_1, y_0) = f(x_3, x_2, x_1, x_0)$
in which $x_0, x_1, x_2, x_3, y_0, y_1 \in \{0,1\}$

(represented by a truth table, which lists the value of (y_1, y_0) for all 16 possible inputs)



A logic circuit with 7 AND (NAND) gates that exactly implements f

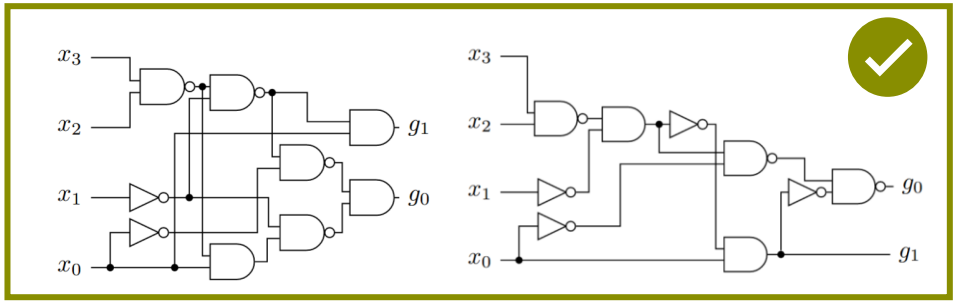
The Challenge: Preserving Logical Equivalence

Boolean function f

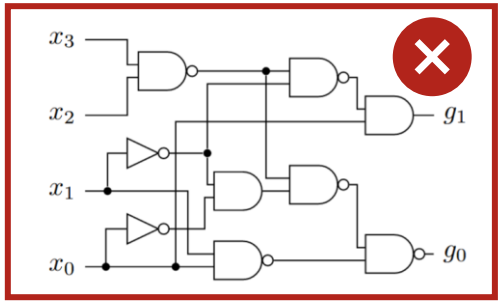
Must be **exactly** equivalent (i.e., without a single bit of error!)

x_3	x_2	x_1	x_0	y_1	y_0	x_3	x_2	x_1	x_0	y_1	y_0
0	0	0	0	0	1	1	0	0	0	0	1
0	0	0	1	0	0	1	0	0	1	0	0
0	0	1	0	0	0	1	0	1	0	0	0
0	0	1	1	1	1	1	0	1	1	1	1
0	1	0	0	0	1	1	1	0	0	0	0
0	1	0	1	0	0	1	1	0	1	1	1
0	1	1	0	0	0	1	1	1	0	0	0
0	1	1	1	1	1	1	1	1	1	1	1

Logic circuit g



Outputs match for all 16 possible inputs



Outputs not exactly match

Motivation: Stepwise decomposition of a property

Typical Gen AI models (e.g., Transformer) are step-by-step sequence generators.

If we want to generate a sequence $s_1, s_2, \dots, s_N$ with certain property $F(s_1, \dots, s_N)$ strictly holds, we can decompose F as N "cutoff properties":

- $F(s_1)$
- $F(s_1, s_2)$
- $F(s_1, s_2, s_3)$
- ...
- $F(s_1, s_2, \dots, s_N)$

hold → hold → hold → hold

Step-by-step "partial feasible" construction towards a full feasible solution

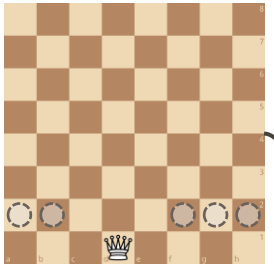
In this work, similar to the 8-queen problem, we incrementally construct a logic circuit with "cutoff properties" that preserve equivalence

Example: the 8-queen problem

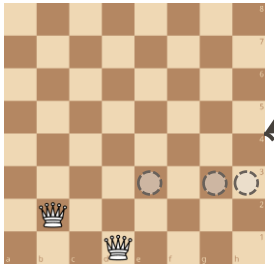
Desired property: all the 8 queens do not attack each other

$F(a, b, c, \dots)$: queens at a, b, c, … do not attack each other

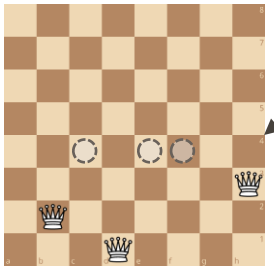
Step 1: $F(d1)$ holds
Queen at d1 will not attack each other
Easy to check
$F(d1, a2), F(d1, b2), F(d1, f2), F(d1, g2), F(d1, h2)$ hold
(valid choices $S_1 = \{a2, b2, f2, g2, h2\}$)
Choose $b2 \in S_1$



Step 2: $F(d1, b2)$ holds
Queens at d1, b2 will not attack each other
Easy to check
$F(d1, b2, e3), F(d1, b2, g3), F(d1, b2, h3)$ hold
(valid choices $S_2 = \{e3, g3, h3\}$)
Choose $h3 \in S_2$

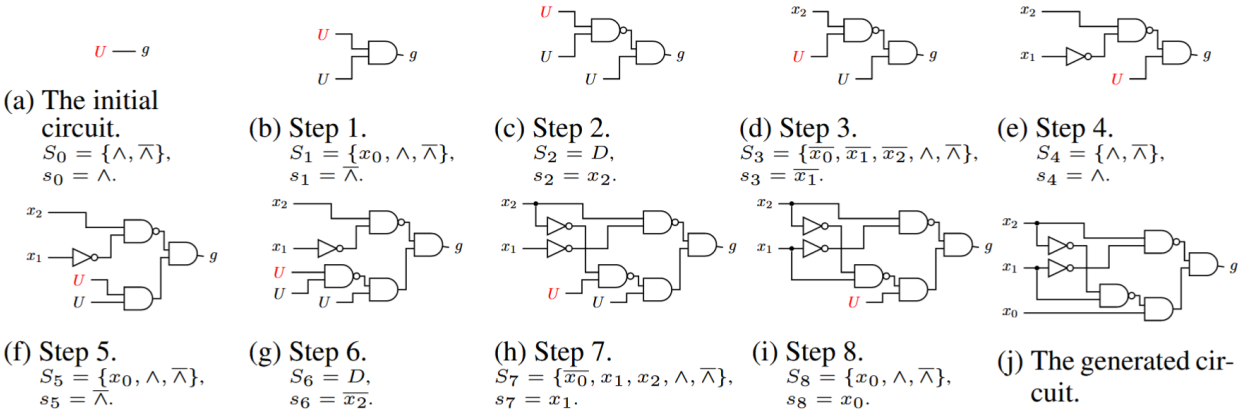


Step 3: $F(d1, b2, h3)$ holds
Queens at d1, b2, h3 will not attack each other
Easy to check
$F(d1, b2, h3, c4), F(d1, b2, h3, e4), F(d1, b2, h3, f4)$ hold
(valid choices $S_3 = \{c4, e4, f4\}$)
…



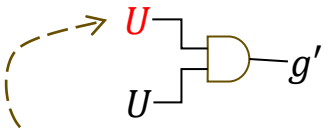
A Sequential Representation of Circuits with “Cutoff Properties”

- Construct from outputs to inputs to allow equivalence validation throughout the construction process.
- Initialize a circuit by a wildcard node (U)
- In each step, refine the circuit by replacing a wildcard node with a new gate / input
  - All the inputs of a new gate will be initialized to wildcard nodes
- When multiple wildcard node exist, follow a fixed order to replace them
  - Prioritizes those with the largest distance from the output
  - Prioritizes the left child of a gate over the right one



Computation of Valid Moves S_t

- Attempt to replace U by each possible input / gate
- Compute the output of the current circuit
- If no equivalence conflict occurs, then add it to S_t
  - U (unknown) will not conflict with any output
- This process can be done efficiently with cache and matrix operation



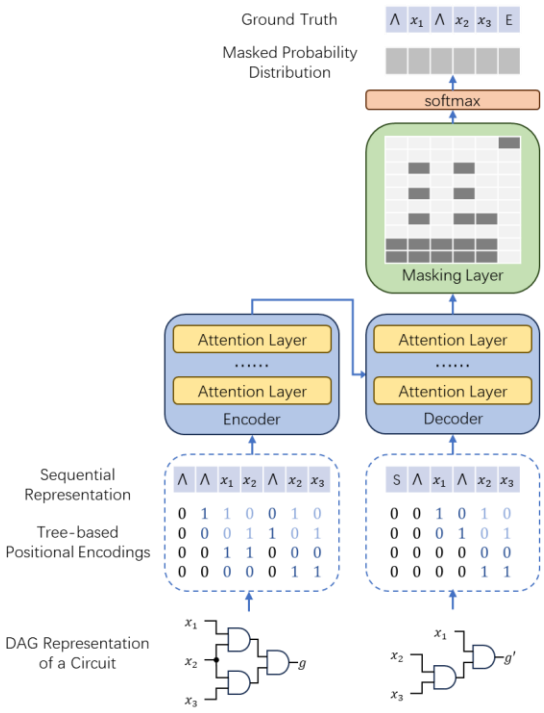
Replace U with each possible input / gate to see whether any equivalence conflict occurs

x_2	x_1	x_0	f	The value of g' when U is replaced by							
				x_0	$\overline{x_0}$	x_1	$\overline{x_1}$	x_2	$\overline{x_2}$	$\wedge$	$\overline{\wedge}$
0	0	0	**0**	0	U	0	U	0	U	U	U
0	0	1	**1**	U	0	0	U	0	U	U	U
0	1	0	**0**	0	U	U	0	0	U	U	U
0	1	1	**0**	U	0	U	0	0	U	U	U
1	0	0	**0**	0	U	0	U	U	0	U	U
1	0	1	**0**	U	0	0	U	U	0	U	U
1	1	0	**0**	0	U	U	0	U	0	U	U
1	1	1	**1**	U	0	U	0	U	0	U	U

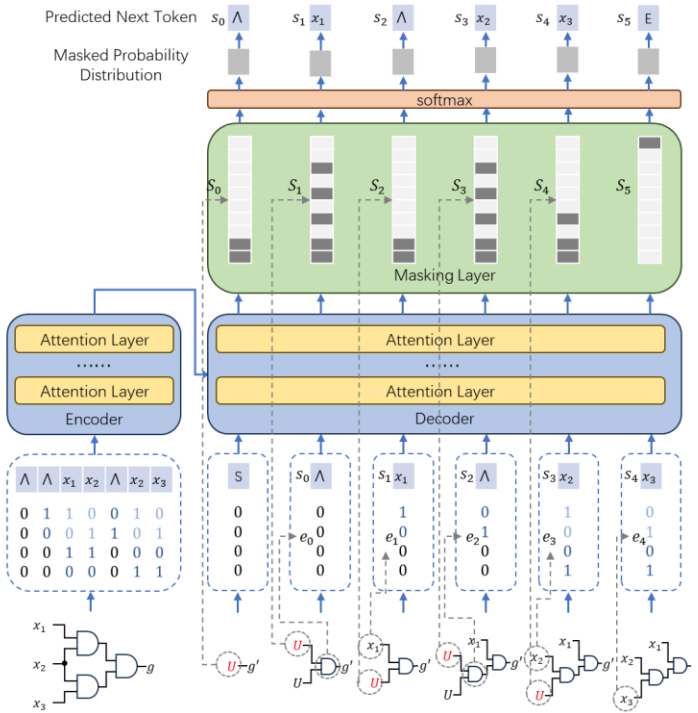
$S_t = \{x_0, \wedge, \overline{\wedge}\}$

The Circuit Transformer

- The Boolean function is encoded by the Transformer encoder
- A masking layer is added before the softmax layer of the Transformer decoder
  - Only allows tokens in valid move S_t to have positive possibilities



The Training Stage



The Inference Stage

Experiments

- A Circuit Transformer with 88M parameters is trained on 15M random generated circuits
  - Circuit size: 8-input, 2-output (can specify $1.34 \cdot 10^{154}$ different circuits)
- Task: generate equivalent yet more compact forms of input circuits
- Results
  - Zero violation of equivalence constraints
  - Optimization performance close to ground truth (even better with MCTS enabled)

In distribution ↓ Out of distribution ↓

Methods	Random circuits		IWLS FFWs	
	Unsuccessful cases	Avg. size	Unsuccessful cases	Avg. size
Boolean Chain	5.07% (5.07%)	15.25	11.36% (11.26%)	17.24
Boolean Chain (beam size = 16)	2.16% (2.16%)	14.89	6.34% (6.29%)	17.15
Boolean Chain (beam size = 128)	1.91% (1.91%)	14.87	5.97% (5.94%)	17.15
AIGER	4.32% (4.32%)	15.14	8.35% (7.77%)	17.19
AIGER (beam size = 16)	1.85% (1.85%)	14.87	4.62% (4.37%)	17.12
AIGER (beam size = 128)	1.71% (1.71%)	14.86	4.24% (3.99%)	17.12
Circuit Transformer w/o TPE	2.14% (0%)	15.02	6.63% (0%)	17.33
Circuit Transformer	1.14% (0%)	14.79	4.76% (0%)	17.17
Circuit Transformer ($K = 10$)	0.20% (0%)	14.02	2.83% (0%)	16.92
Circuit Transformer ($K = 100$)	**0.17%** (0%)	**13.73**	**2.63%** (0%)	**16.73**
Resyn2 (ground truth for training)	/	14.56	/	16.82

Better performance with MCTS enabled

Zero violation of equivalence constraints

References

- Willard, B. T., & Louf, R. (2023). Efficient Guided Generation for Large Language Models. *arXiv preprint arXiv:2307.09702*.
- Earley, J. (1970). An efficient context-free parsing algorithm. *Communications of the ACM*, 13(2), 94–102.
- Li, X., Li, X., Chen, L., Zhang, X., Yuan, M., & Wang, J. (2025). Circuit Transformer: A Transformer That Preserves Logical Equivalence. *Proceedings of the International Conference on Learning Representations (ICLR 2025)*. arXiv:2403.13838.

Google Developer Experts

Thank you!

Xihan Li, Ph.D.
Google Developer Expert in AI
Incoming Assistant Professor, Fudan University
https://snowkylin.github.io