Logic Synthesis with Generative Deep Neural Networks

Xihan Li
University College London
London, UK
xihan.li@cs.ucl.ac.uk

Xing Li
Huawei Noah's Ark Lab
Hong Kong, China
li.xing2@huawei.com

Lei Chen
Huawei Noah's Ark Lab
Hong Kong, China
lc.leichen@huawei.com

Xing Zhang
Huawei Noah's Ark Lab
Hong Kong, China
zhangxing85@huawei.com

Mingxuan Yuan
Huawei Noah's Ark Lab
Hong Kong, China
Yuan.Mingxuan@huawei.com

Jun Wang
University College London
London, UK
jun.wang@cs.ucl.ac.uk

ABSTRACT

While deep learning has achieved significant success in various domains, such as board games and conversational agents, its application to logic circuit design has been limited due to complex constraints and strict feasibility requirement. However, a recent generative deep neural model, "Circuit Transformer", has shown promise in this area by enabling equivalence-preserving circuit transformation on a small scale. In this paper, we introduce a logic synthesis rewriting operator based on the Circuit Transformer model, named "`ctrw`" (Circuit Transformer Rewriting), which incorporates the following techniques: (1) a two-stage training scheme for the Circuit Transformer tailored for logic synthesis, with iterative improvement of optimality through self-improvement training; (2) integration of the Circuit Transformer with state-of-the-art rewriting techniques to address scalability issues, allowing for guided DAG-aware rewriting. Experimental results on the IWLS 2023 contest benchmark demonstrate the effectiveness of our proposed rewriting methods.

CCS CONCEPTS

• **Hardware** → **Electronic design automation**; • **Computing methodologies** → **Artificial intelligence**.

KEYWORDS

Generative AI, Circuit Transformer, Logic Synthesis

1 INTRODUCTION

In recent years, deep learning has demonstrated rapid advancements and remarkable success across various domains. Two particularly notable achievements include the triumph of deep reinforcement learning in board games, exemplified by AlphaGo's victory in 2016 [19], and the success of generative neural networks in conversational agents, illustrated by the release of ChatGPT in 2022 [16]. In domains such as board games and conversational agents, there is a consensus that deep learning-based solutions have established a "dominance", significantly surpassing traditional methods in terms of performance and capability.

However, in fields such as logic circuit design, despite numerous publications on deep-learning related research, deep learning has not yet achieved a dominant position. Traditional methods still play a central role due to domain-specific challenges where deep learning has inherent limitations. For logic circuits, one of the core issues is the need to adhere to strict logical constraints without

any margin for error [8]. For instance, a synthesized circuit must be exactly equivalent to the original, while deep learning, being fundamentally a regression technique, struggles to eliminate errors entirely. This conflict significantly limits the effectiveness of deep learning in logic circuit design. Consequently, most deep learning research in this area focuses on augmenting traditional methods that ensure constraint satisfaction, rather than directly constructing logic circuits. [2, 4–7, 12, 15, 23]

In 2024, the introduction of the Circuit Transformer [11] marked a breakthrough in addressing the constraint issues inherent in logic circuit design. By employing a specially designed encoding-decoding process, the Circuit Transformer facilitates the generation of And-Inverter Graphs (AIGs) in a manner analogous to large language models such as ChatGPT. Crucially, this model precisely preserves equivalence constraints. Although the current implementation is limited to processing small circuits comprising up to approximately 30 gates on a consumer-grade computer, it demonstrates the capability to construct feasible circuits from scratch using a fully neural approach. This achievement indicates that the core issue of strict logical constraints in deep learning for circuit design can indeed be overcome, demonstrating a potential paradigm shift in this field.

Given that the core issue has been resolved, this paper takes a further step with Circuit Transformer, demonstrating how this neural model can serve as the foundation for rewriting in logic synthesis. To train Circuit Transformer specifically for logic synthesis, we propose a two-stage scheme. In the first stage, the model is trained in a supervised manner to emulate traditional synthesis tools. In the second stage, we iteratively enhance the model's optimality using Monte-Carlo Tree Search (MCTS). This approach is analogous to AlphaGo's training process, which first learned from human expert moves and then improved through iterative training on self-play datasets. This method allows Circuit Transformer to autonomously explore and exploit new optimization strategies without relying on human expertise.

Then, to address the scalability issue, we integrate Circuit Transformer with a recent rewrite framework based on fanout-free windows [24], which are sub-structures of circuits containing one or more outputs. Instead of optimizing the entire circuit, Circuit Transformer focuses on optimizing these fanout-free windows, which are typically small in size. This approach leverages the strengths of Circuit Transformer to handle these smaller, manageable portions effectively. From a sub-graph rewriting perspective, compared to traditional methods like enumerating all NPN classes with a cut size
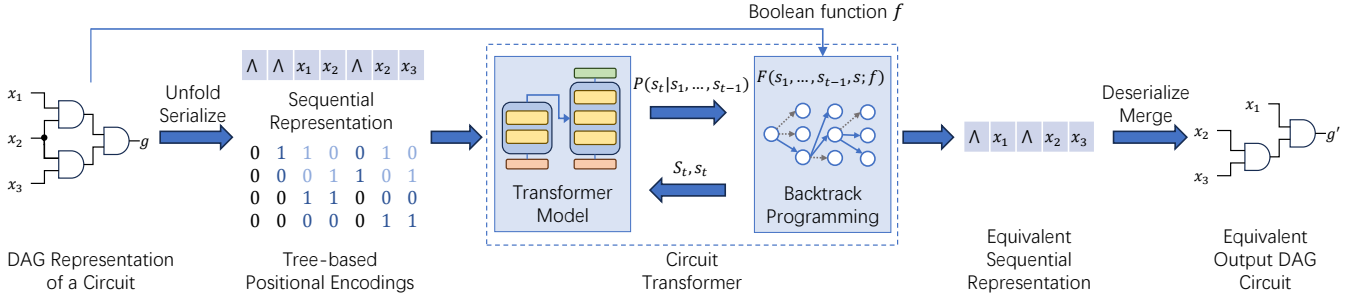
Figure 1: The pipeline of Circuit Transformer that transforms a circuit to a strictly equivalent one.

of 4 [14] or exact synthesis with a cut size of 6 [18], Circuit Transformer offers greater scalability. It can process larger sub-graphs with a window size of 8 and multiple outputs. This capability makes it particularly well-suited for rewriting, as considering larger sub-graphs holds more potential for creating optimized networks.

Finally, we demonstrate that Circuit Transformer can be adapted for Directed Acyclic Graph (DAG) aware rewriting. Benefiting from a Markov Decision Process (MDP) formulation, Circuit Transformer can incorporate context information (i.e., nodes outside the fanout-free window) during the generation, which allows the model to favor the reuse of existing nodes. Specifically, when a node is generated and identified as equivalent to an existing node in the context, the immediate reward at that step is refined as if the generated node were merged with the existing one. This refinement ensures that the cumulative reward of the MDP accurately reflects the number of added nodes after the merging process. Consequently, Monte-Carlo Tree Search (MCTS) is guided by the reward to generate circuits that may not be the smallest in size but result in a more optimized overall circuit after node merging.

Combining all the techniques proposed above, we developed a new rewrite operator named `ctrw` (Circuit Transformer Rewriting). Despite currently training a small Circuit Transformer on a consumer-grade computer, `ctrw` has demonstrated its effectiveness on the IWLS 2023 contest benchmark. Future work will focus on scaling up the Circuit Transformer model using large-scale AI platforms and enhancing both its efficiency and performance with state-of-the-art AI techniques.

2 PRELIMINARIES

2.1 And-Inverter Graph and Fanout-Free Window

An And-Inverter Graph (AIG) is a data structure to model combinational logic circuits [1]. Let $G = (V, E, PI, PO)$ be an AIG, where (V, E) is a directed acyclic graph, PI is the set of primary input vertices, and PO is the set of primary output vertices. Each vertex $v \in V$ represents an And gate and $v \in PI$ represents a primary input. Edges represent wires and can either be regular or complemented.

Given a set I of k input vertices , a k-input fanout-free window (FFW) $G = (V, E, I, O)$ is also an AIG, where I are the input vertices and $O \subset V$ are the output vertices. A fanout-free window should satisfy following fanout-free rules,

(1) For each vertex $v \in V$, $FI(v) \subset V$.
(2) For each vertex $v \in V\ O$, $FO(v) \subset V$.

We refer to [24] for more details.

2.2 Transformer and Circuit Transformer Model

The Transformer model [21] is a deep learning architecture that revolutionized artificial intelligence, particularly with the emergence of large language models. It typically runs as a generative model that accepts a sequence of tokens (words) as input, and recurrently predict the next token (word) to be generated. Unlike traditional sequential models, transformers process entire sequences simultaneously, allowing for efficient parallelization and capturing long-range dependencies. This architecture, coupled with innovations like multi-head attention and positional encoding, enables transformers to excel in diverse tasks such as natural language processing and image recognition.

The Circuit Transformer model [11], as the name suggests, is a variation of Transformer model for logic circuits. It consists of an encoding scheme that transform an And-Inverter Graph to a trajectory that can be efficiently processed by Transformer models, and a decoding scheme that guarantee the generated circuit to adheres to specified equivalence constraints. It runs typically in a generative way that accepts an encoded AIG as input, and recurrently predict the next node (AND gate or PI), thereby constructing an output AIG step by step. Its pipeline is shown in Figure 1 and the generation process is shown in Algorithm 1.

2.3 Markov Decision Process and Monte-Carlo Tree Search

Markov Decision Processes (MDPs) is a mathematical framework for modeling decision-making problems, in which an agent interacts with an environment over a series of discrete time steps. At each step, the agent chooses an action based on the current state of the environment, leading to a transition to a new state and receiving a reward. By solving MDPs, agents can learn optimal policies (i.e., optimal mappings from state to action) that maximize cumulative rewards over time.

Monte Carlo Tree Search (MCTS) is a heuristic search algorithm renowned for its effectiveness in decision-making under uncertainty. Unlike traditional search methods, MCTS iteratively builds

---

Algorithm 1 Processing an AIG with Circuit Transformer

Input: An input AIG G to be processed, an optional set of mapping C for the ith primary output of the generate AIG, a trained Circuit Transformer $P_{CT}(g_n|g_{1:n-1}, G_{enc}, \{TT_i\})$

Output: A processed AIG G'.

Encode G as G_{enc} following Section 3.1 of [11]

Initialize $G_{dec} \leftarrow [\,]$

while $t = 0, 1, 2, \ldots$ **do**

    Compute a probability distribution of nodes

$$p_t \leftarrow P_{CT}(g_t|G_{dec}, G_{enc}, \{TT_i\})$$

    Compute the tth node $g_t \leftarrow \arg\max p_t$

    If g_t = EOS **then break**

    Add g_t to G_{dec}

end while

Decode G_{dec} as G' following Section 3.3 of [11]

Return G'

---

a search tree by sampling potential trajectories and dynamically balancing exploration and exploitation. Through repeated simulations, MCTS efficiently navigates large decision spaces, finding optimal solutions in domains with complex branching factors and uncertain outcomes.

2.4 Rewriting and DAG-aware Rewriting

Rewriting is a powerful and widely used area-oriented optimization method that iteratively selects and greedily replaces small-scale sub-graphs with more compact structures. The sub-graphs can be either single-output (k-input cuts [14]) or multi-output (k-input fanout-free window [24]).

The idea of DAG-aware rewriting is to replace the aforementioned small-scale subgraphs with ones that can make the whole graph more compact by utilizing existing logic. Such a replacement sub-graph is not necessary to be smaller than the original one. Existing approaches compute multiple candidate replacements offline [14] or on-the-fly [18], then enumerate each candidate during the rewriting process, and select the one with best gain to replace the original sub-graph. Information about existing logic are not utilized when the candidate replacements are generated.

3 METHODS

3.1 Train a Circuit Transformer for Logic Synthesis

In this section, we train a Circuit Transformer model to perform logic synthesis on small-scale AIGs in a fully generative manner. The input to the model is an encoded AIG that needs optimization. The neural model then generates the synthesized AIG step by step by recurrently predicting the next node of the AIG. The output AIG is equivalent to the original but expected to be more compact in size.

To train the Circuit Transformer model for logic synthesis, we propose a two-stage training scheme. In the first stage, the model undergoes supervised training using pairs of data in the form <original circuit, optimized circuit>. The original circuits are randomly generated, while the optimized circuits are produced using conventional logic synthesis libraries, such as `resyn2` in ABC [3]. This approach

---

Algorithm 2 Random generation of a k-input, l-output AIG

Input: Number of input k, number of output l, number of steps M_{step}.

Output: A randomly generated AIG with k inputs and l outputs.

$S \leftarrow \{I_0, I_1, \ldots, I_{k-1}\}$

for $i = 1, 2, \ldots, M_{step}$ **do**

    Create an AND node a_i

    Randomly sample two nodes $c_0, c_1 \in S$ without replacement

    Set the first input of a_i as c_0 or $\overline{c_0}$ randomly

    Set the second input of a_i as c_1 or $\overline{c_1}$ randomly

    Add a_i to S

end for

Return an AIG with $I_0, I_1, \ldots, I_{k-1}$ as primary inputs and $a_{M-l+1}, a_{M-l+2}, \ldots, a_M$ as primary outputs.

---

initializes the Circuit Transformer by emulating the strategies used by traditional synthesis tools.

To generate a k-input, l-output AIG in a random manner, we are inspired by the Boolean chain [9], starting by initializing a set of nodes $S = \{I_0, I_1, \ldots, I_{k-1}\}$, where $I_0, I_1, \ldots, I_{k-1}$ are input nodes. We then iteratively create AND nodes a_i. For each a_i, we randomly select two nodes $c_0, c_1 \in S$ as its inputs, setting the connecting wires to be either regular or inverted at random. The AND node a_i is subsequently added to S. After M iterations, the last l nodes are designated as the primary outputs of the generated AIG. The detailed process is outlined in Algorithm 2.

Canonicalization techniques [24] are applied to filter out AIGs with pre-existing canonicalizations in the dataset. This ensures that each randomly generated AIG in the dataset has a unique structure, thereby enhancing dataset diversity.

It is worth noting that, although realistic data is generally considered superior to synthetic data for training, we found it challenging to train an effective model using cuts or fanout-free windows extracted from realistic circuits. We hypothesize that this difficulty arises because the differences between original and optimized circuits are typically minimal in real cuts or fanout-free windows, making it hard for the model to learn effective node-reduction strategies.

3.2 Iterative Self-Improvement Training

Given an initial Circuit Transformer, the second stage focuses on iteratively enhancing the model's ability to generate more compact circuits. To achieve this, we attach an immediate reward function to each step of the circuit generation process described in Algorithm 1. This transforms the stepwise process into a Markov Decision Process (MDP), where the state, action, and reward at step t are defined as follows:

- State: the generated token sequence $g_1, \ldots, g_t$
- Action: all the valid choices of g_{t+1}.
- Immediate Reward:

$$R(g_1, \ldots, g_t, g) = \Delta + \begin{cases} 1, & g = \wedge \text{ or } g = \overline{\wedge} \\ 0, & \text{otherwise} \end{cases} \quad (1)$$

    in which Δ reflects the refinement due to equivalent node merging. An example can be found in Figure 4a.

In this way, the cumulative reward is the negative of number of AND nodes in the generated AIG, and the Circuit Transformer
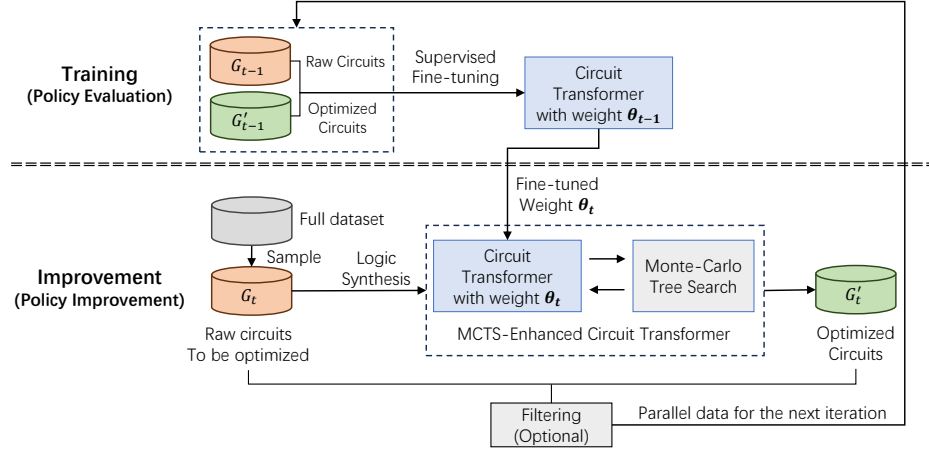
Figure 2: The process of Iterative Self-Improvement Training. Two stages are included in a single iteration: (1) Training: the Circuit Transformer is supervisedly fine-tuned by circuit pairs $< G_{t-1}, G'_{t-1} >$, with its weight updated from θ_{t-1} to θ_t; (2) Improvement: a batch of raw circuits G_t is sampled from the full dataset, and processed by MCTS-enhanced Circuit Transformer with weight θ_t to generate further optimized circuits G'_t.

model serves as a policy, mapping from state $g_1, \ldots, g_t$ to action g_{t+1}. Consequently, the task of logic synthesis, with its core objective of minimizing the number of AND nodes, can be reformulated as solving this MDP.

To solve this MDP, we need an effective method to iteratively improve the policy, as represented by the Circuit Transformer. While policies in MDPs can often be enhanced using deep reinforcement learning (DRL) algorithms such as Q-learning or policy gradient methods, the unique features of the Transformer-based architecture prevent straightforward application of these algorithms. Our observations indicate that directly applying online DRL algorithms to this MDP results in non-convergence, and offline DRL methods do not yield significant improvements either. This behavior may be attributed to the extreme sensitivity of DRL algorithms to network architectures and the substantial non-stationarity during training, as discussed in [10].

To stabilize the training process, first, we propose leveraging Monte-Carlo Tree Search (MCTS) to improve the policy. When deciding on the t-th node g_t, instead of merely selecting the node with the highest probability from the Circuit Transformer, we build a search tree to evaluate multiple choices of g_t (i.e., actions in the MDP) through simulation. The Circuit Transformer model acts as a guide to prioritize exploration. The detailed process is outlined in Algorithm 3.

Next, we introduce Iterative Self-improvement Training to iteratively fine-tune the Circuit Transformer model. In each iteration, the Circuit Transformer is first fine-tuned on data pairs <original circuits, optimized circuits> in a supervised manner. Then, we randomly sample a set of circuits from the complete dataset and process them through the MCTS-enhanced Circuit Transformer to generate further optimized circuits. These sampled circuits and their corresponding enhanced optimized circuits then serve as the data pairs for fine-tuning in the next iteration. The detailed iterative training process is depicted in Figure 2 and Algorithm 4.

3.3 Cooperate Circuit Transformer with Fanout-Free Window Rewriting

While the Circuit Transformer model we trained in the previous section can already perform end-to-end logic synthesis with decent performance, it is of limited scalability. In exchange for the decoding efficiency of Transformer models, Circuit Transformer's encoding scheme allows "unfolding" with significant redundancy, which hinders it to process very large circuits (as the encoded sequence will be prohibitively long). On a customer-grade computer equipped with a single high-end GPU, we are able to train a Circuit Transformer that can process AIGs of up to around 30 AND nodes. Even if resorting to large-scale AI training clusters (which we are actually working on), we would not expect Circuit Transformer to process AIGs with thousands of nodes in a single shot.

Therefore, to address the scalability issue, we also resort to so-called peephole optimization, which partitions a circuit into small sub-circuits that can be replaced by a more compact implementation. More specifically, we leverage the fanout-free window (FFW) rewriting framework proposed in [24], and apply our trained Circuit Transformer with MCTS enhancement to replace each fanout-free window by a more compact one. We name such a rewrite operator as "*Circuit Transformer Rewriting*" (`ctrw`). A filter will be applied to candidate FFWs, to ensure that the circuits are within the trained model's capacity after encoding.

Moreover, Circuit Transformer enables additional flexibility for the rewriting process. For the partition-and-replacement scheme of rewriting frameworks, an important observation is that, while the new implementation should not change the original functionality of the whole circuit, the new implementation itself is not necessary to be logical equivalent to the sub-circuit it replaces [17]. An example is shown in Figure 3. Traditional methods like offline replacement computation [14] or on-the-fly exact synthesis [18] have to preserve such logical equivalence between sub-circuits, resulting in a loss

---

Algorithm 3 Logic synthesis with MCTS-enhanced Circuit Transformer

---

Input: An input AIG G to be processed, an optional truth table TT_i for the ith primary output of the generate AIG, a trained Circuit Transformer $P_{CT}(g_n|g_{1:n-1}, G_{enc}, \{TT_i\})$, number of MCTS steps M_{step}, number of MCTS playouts per step $M_{playout}$

Output: A processed AIG G'.

  procedure PUCT(an MCTS node x)
    for a in x's all child nodes **do**
      $s_a \leftarrow \frac{a.\text{total\_value}}{a.\text{visited}} + a.\text{prob}\sqrt{\frac{x.\text{visited}}{1+a.\text{visited}}}$
    end for
    Return $\arg\max_a s_a$
  end procedure
  procedure MCTS(P_{CT})   ▷ See [20] for details
    Create a root MCTS node r
    for $i = 1, 2, \ldots, M_{playout}$ **do**
      (Selection) Starting from r, iteratively selects a child node via PUCT algorithm until reaching a leaf node l.
      (Expansion) Evaluate l via P_{CT} (i.e., create all child nodes a for l, and assign $a.\text{prob}$ for each child via P_{CT}), and select a child node via PUCT.
      (Simulation) Run the generation process in Algorithm 1 via P_{CT} until reaching EOS or the maximum #(iter), and get the cumulative reward v
      (Backpropagation) Update the "visited" and "total_value" attributes of the MCTS nodes from p to r in a backward pass with v.
    end for
    Return An action of r in which the maximum cumulative reward appeared in the corresponding branch.
  end procedure
  Encode G as G_{enc} following Section 3.1 of [11]
  Initialize $G_{dec} = [\,]$
  while $t = 0, 1, 2, \ldots$ **do**
    if $t < M_{step}$ **then**
      Compute the t-th node $g_t \leftarrow$ MCTS(P_{CT})
    else
      $g_t \leftarrow \arg\max P_{CT}(g_t|G_{dec}, G_{enc}, \{TT_i\})$
    end if
    If $g_t = $ EOS **then break**
    Add g_t to G_{dec}
  end while
  Decode G_{dec} as G' following Section 3.3 of [11]
  Return G'

Algorithm 4 Iterative improvement training process

Input: The Circuit Transformer model P_{CT} to be fine-tuned, number of iterations M_{iter}, dataset size per iteration M_{size}, a pre-generated dataset D, number of MCTS playouts $M_{playout}$

Output: An enhanced Circuit Transformer model.

  $P_{CT}^0 \leftarrow P_{CT}$
  for $i = 0, 1, \ldots, M_{iter} - 1$ **do**
    $D_i \leftarrow [\,]$
    while $|D_i| < M_{size}$ **do**
      Sample $(G, G') \in D$
      Do a random NPNP transformation to G.
      Run Algorithm 1 with P_{CT}^i on G to get G_1''
      Run Algorithm 3 with P_{CT}^i on G to get enhanced AIG G_2''
      if $|G_2''| < |G_1''|$ **then**
        Add (G, G_2'') to D_i
      end if
    end while
    Fine-tune P_{CT}^i on dataset D_i to get P_{CT}^{i+1}
  end for
  Return $P_{CT}^{M_{iter}-1}$.

(a) Two AIGs which are not equivalent.

(b) The first AIG, which serves as a sub-graph in this case, can be replaced by the second one.

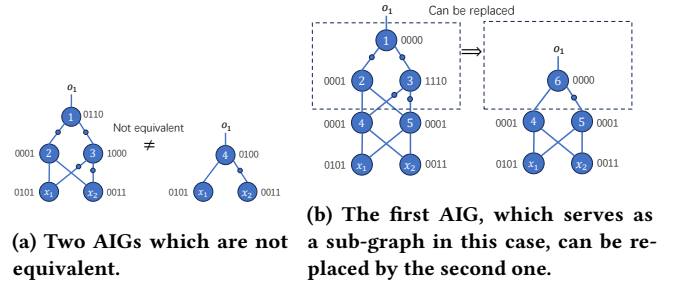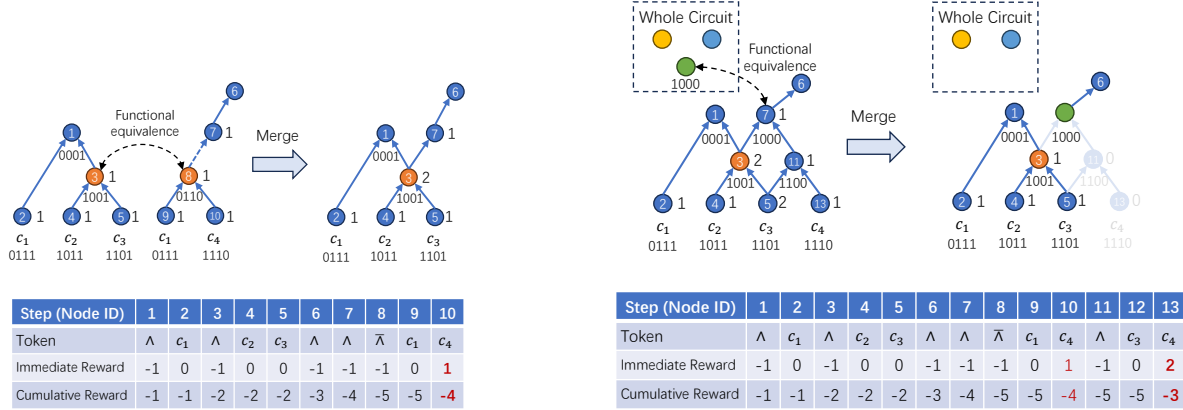**Figure 3: An example showing how a sub-circuit can be replaced by a non-equivalent one, without changing the functionality of the whole circuit.**

of flexibility. A recent study [17] allows such kind of flexibility via a specific encoding of Quantified Boolean Formulas, and Circuit Transformer provides a more straightforward approach with its capability of preserving arbitrary equivalence constraints.

To illustrate, we first introduce how a Circuit Transformer preserves the equivalence between input and output circuits in the previous section. For a circuit with k inputs and l outputs, this is done by specifying $2^k \times l$ constraints so as to fix the truth table of the outputs. For example, for the first AIG in Figure 3a consisting of three AND nodes, when feeding it into a Circuit Transformer to generate a new equivalent AIG, four constraints will be specified: $(x_0 = 0, x_1 = 0 \rightarrow o_1 = 0)$, $(x_0 = 1, x_1 = 0 \rightarrow o_1 = 1)$, $(x_0 = 0, x_1 = 1 \rightarrow o_1 = 1)$, $(x_0 = 1, x_1 = 1 \rightarrow o_1 = 0)$.

Then, when we apply a Circuit Transformer to rewrite a k-input sub-graph within a larger AIG of K inputs, we specify the constraints in a more "globalized" way. That is, we resort to the truth table of the input and output nodes, with respect to the larger AIG (so the size of the truth tables are 2^K instead of 2^k). For example, for the first sub-graph in Figure 3b within a 5-nodes AIG, we utilize the truth table of two outputs (node 4 and 5) and one inputs (node 1) for the whole graph, and specify two constraints for the Circuit Transformer: $(x_0 = 0, x_1 = 0 \rightarrow o_1 = 0)$ and $(x_0 = 1, x_1 = 1 \rightarrow o_1 = 0)$. In such a way, we introduce additional flexibility by reducing the number of equivalence constraints from four to two, leaving the other two input cases ($x_0 = 1, x_1 = 0$ and $x_0 = 0, x_1 = 1$) as don't cares. Once a Circuit Transformer generates an AIG satisfying these two constraints, it can replace the original sub-graph without changing the whole circuit, as the truth table of the output nodes within the whole graph are preserved.

It also worth attention that cycles may appear when replacing a multi-output sub-circuit by a new one, which is also observed in [17]. For our method, we simply check whether a cycle appears after each replacement, and revert the change if a cycle is detected.

Step (Node ID)	1	2	3	4	5	6	7	8	9	10
Token	$\wedge$	c_1	$\wedge$	c_2	c_3	$\wedge$	$\wedge$	$\bar{\wedge}$	c_1	c_4
Immediate Reward	-1	0	-1	0	0	-1	-1	-1	0	**1**
Cumulative Reward	-1	-1	-2	-2	-2	-3	-4	-5	-5	**-4**

(a) In step 10, the functionality of node 8 is fully determined ($c_1 \wedge c_4$), which is functionally equivalent to node 3 ($c_2 \wedge c_3$). Therefore, the two nodes are merged and the number of AND nodes is reduced by one, which is added to the reward of step 10. The cumulative reward up to step 10 is -4, which correctly reflects the number of added AND nodes after merging (node 1, 3, 6, 7).

Step (Node ID)	1	2	3	4	5	6	7	8	9	10	11	12	13
Token	$\wedge$	c_1	$\wedge$	c_2	c_3	$\wedge$	$\wedge$	$\bar{\wedge}$	c_1	c_4	$\wedge$	c_3	c_4
Immediate Reward	-1	0	-1	0	0	-1	-1	-1	0	1	-1	0	**2**
Cumulative Reward	-1	-1	-2	-2	-2	-3	-4	-5	-5	-4	-5	-5	**-3**

(b) In step 13, the functionally of node 11 (truth table: 1100) and node 7 (truth table: 1000) are determined, and node 7 is identified to be functionally equivalent to the green node in the whole circuit. Therefore node 7 is dereferenced as it is replaced by the green node, reducing the number of AND nodes by two (node 7 and 11), which is added to the reward of step 13. The cumulative reward up to step 13 is -3, which correctly reflects the number of added AND nodes after the DAG-aware merging (node 1, 3, 6).

Figure 4: Examples showing how the reward is refined in the sequential generation process, so that the cumulative reward correctly reflects the total number of added AND nodes.

3.4 Circuit Transformer for Guided DAG-aware Rewriting

The idea of DAG-aware rewriting bases on another important observation that, while the aim of rewriting is to make the whole circuit more compact with less nodes, the new implementation itself is not necessary to be more compact than the sub-circuit it replaces. This is because the new implementation may heavily reuse already existing logic in the whole circuit, which will not be counted after the replacement. Previous works [14, 18] implement DAG-aware rewriting via generating multiple candidate circuits for a single replacement, and select the one that leads to a minimal number of nodes of the whole circuit after replacement. Such methods are less "guided" as the structure of the whole circuit is not considered during the generation of candidate circuits.

Circuit Transformer, however, is able to consider such context information during the MCTS enhancement described in Algorithm 3, so as to perform a "guided search" that favor the reuse of existing logic, and minimize the number of nodes of the whole circuit. This is done by refining the definition of reward in the MDP proposed in Section 3.1, to correctly reflect the node reduction of the whole circuit after replacement, by taking existing nodes' reuse into consideration. More specifically, we first de-reference the sub-graph to be replaced, then during the sequential generation process in Algorithm 3, when we found that the functionality of a generated AND node can be fully determined (i.e., no nodes it depends on are still waiting to be generated), we will check whether the node is functionally equivalent to an existing node in the whole circuit. If such a node exists, we will do a "mock de-reference" of the generated node to see how many nodes can be reduced by replacing this generated node with the existing one. Then, we add the number

of "mock de-referenced" nodes to the reward in the current step. In such a way, the cumulative reward of the MDP will correctly reflect the influence of node reuse, and MCTS will be directed to generate circuits that may not be the most compact in size, but leads to more reduced node of the whole circuit after replacement. An example is shown in Figure 4b. We argue such a "DAG-aware generation process" can be more efficient than traditional techniques in finding an optimal circuit structure for replacement.

4 EXPERIMENTS

To employ the mature development ecosystem of deep learning, We implement our method fully in Python with TensorFlow. While the inference speed of Circuit Transformer limits the scalability of our implementation on customer-grade computers, it can be significantly scaled and accelerated on modern large-scale AI clusters with thousands of GPUs, which is our ongoing work. Before that, we trained and evaluated a relatively small Circuit Transformer on a customer-grade computer with dual NVIDIA RTX 4090 GPUs, and release preliminary results in this section to show the effectiveness of our proposed rewriting method.

To train the Circuit Transformer model, we build a dataset containing 12 million randomly generated 8-input, 2-output AIGs. The supervised signal (i.e., the synthesized circuits) are generated by the resyn2 command in ABC [3]. We restrict that the length of the encoded sequence for each AIG should be no more than 200, all the 8 inputs should appear in the AIG, and each AIG has an unique canonicalization.

The details of the Circuit Transformer model are as follows. We employed an encoder-decoder architecture following [21], each

Benchmark		Existing Rewriting Methods						Proposed Rewriting Method												
		Drw Rewriting (abc)			MFFW Rewriting (python)			ctrw w/o Iterative Self-Improvement Training			ctrw			ctrw with MCTS			ctrw with MCTS & guided DAG-aware rewriting		
Name	Size	Size	Improv.	Time	Size	Improv.	Time	Size	Improv.	Time	Size	Improv.	Time	Size	Improv.	Time	Size	Improv.	Time
ex12	12	12	0.00%	0.01	12	0.00%	5.25	12	0.00%	13.55	12	0.00%	16.07	10	16.67%	294.28	10	16.67%	242.95
ex17	17	17	0.00%	0.01	17	0.00%	3.51	17	0.00%	0.80	17	0.00%	0.92	17	0.00%	356.27	16	5.88%	417.22
ex32	33	29	12.12%	0.01	30	9.09%	2.63	31	6.06%	14.65	27	18.18%	16.10	27	18.18%	2618.13	24	27.27%	2748.27
ex23	26	24	7.69%	0.01	24	7.69%	49.33	24	7.69%	27.68	24	7.69%	42.62	24	7.69%	4246.21	22	15.38%	2534.06
ex89	14	13	7.14%	0.01	13	7.14%	4.63	13	7.14%	3.21	13	7.14%	3.00	12	14.29%	686.61	12	14.29%	674.20
ex99	43	41	4.65%	0.01	38	11.63%	3.53	40	6.98%	39.83	35	18.60%	34.15	34	20.93%	3208.47	35	18.60%	3809.46
ex30	28	24	14.29%	0.01	24	14.29%	7.17	25	10.71%	12.41	22	21.43%	16.65	22	21.43%	2050.01	21	25.00%	1942.07
ex07	39	27	30.77%	0.01	27	30.77%	1.63	27	30.77%	30.18	27	30.77%	7.81	27	30.77%	712.62	27	30.77%	691.69
ex96	37	19	48.65%	0.01	19	48.65%	3.58	19	48.65%	9.93	19	48.65%	4.31	19	48.65%	806.41	19	48.65%	659.56
ex41	39	25	35.90%	0.01	20	48.72%	0.91	24	38.46%	6.51	21	46.15%	24.68	21	46.15%	2566.13	20	48.72%	2679.71
ex57	44	40	9.09%	0.01	40	9.09%	158.45	40	9.09%	112.32	40	9.09%	107.29	38	13.64%	16863.31	38	13.64%	11793.32
ex21	48	39	18.75%	0.01	33	31.25%	27.75	38	20.83%	21.39	35	27.08%	37.97	35	27.08%	4062.32	33	31.25%	3692.64
ex53	71	63	11.27%	0.01	53	25.35%	123.37	53	25.35%	208.82	54	23.94%	235.94	53	25.35%	21801.94	51	28.17%	19789.97
ex35	75	57	24.00%	0.01	53	29.33%	11.36	54	28.00%	169.56	41	45.33%	115.18	44	41.33%	9810.85	34	54.67%	7157.42
ex14	53	47	11.32%	0.01	42	20.75%	1.31	48	9.43%	11.65	42	20.75%	24.61	39	26.42%	2566.86	38	28.30%	2577.84
ex49	66	61	7.58%	0.01	60	9.09%	304.63	60	9.09%	249.72	60	9.09%	285.02	61	7.58%	31421.68	56	15.15%	35465.11
ex79	59	46	22.03%	0.01	30	49.15%	0.98	31	47.46%	67.46	30	49.15%	86.21	28	52.54%	6356.74	28	52.54%	5996.43
ex44	81	70	13.58%	0.01	68	16.05%	2.38	67	17.28%	120.17	65	19.75%	169.98	56	30.86%	9831.79	52	35.80%	12344.27
ex95	70	53	24.29%	0.01	42	40.00%	3.13	43	38.57%	203.44	43	38.57%	236.67	43	38.57%	22409.89	40	42.86%	19073.88
ex06	73	69	5.48%	0.01	62	15.07%	65.12	61	16.44%	44.97	60	17.81%	57.12	58	20.55%	8963.9	55	24.66%	7096.53
ex90	95	82	13.68%	0.01	72	24.21%	8.71	80	15.79%	171.38	68	28.42%	142.22	64	32.63%	14827.17	48	49.47%	9050.18
ex04	77	64	16.88%	0.01	63	18.18%	18.95	66	14.29%	101.95	59	23.38%	93.34	53	31.17%	8131.76	49	36.36%	6717.23
Avg. Improv.			15.42%			21.16%			18.55%			23.23%			26.02%			30.19%	

Table 1: Results on 22 small cases of IWLS 2023 contest dataset. Time is measured in seconds.

with 12 attention layers. The embedding width and the size of feed-forward layer are set as 512 and 2048, leading to 88,225,800 total parameters. The implementation is based on [22]. The vocabulary size is 20 (8 POs and AND gate with their inverse, plus [EOS] and [PAD]). Batch size and learning rate are set as 96 and 10^{-3} respectively. The maximum length of the input and output sequence are set as 200 and 100. For the first supervised stage, We trained the model for 1 million batches. For the second iterative enhancement stage, we first set $M_{size} = 10^6, M_{step} = 3, M_{playout} = 10$ for 5 iterations without the filtering condition in line 9 of Algorithm 4, and then set $M_{size} = 10^5, M_{step} = 1, M_{playout} = 10$ for 24 iterations with the filtering condition. Note that all these parameters are selected to be particularly small so as to fit the limited computational capability of the computer we used.

To evaluate the optimization capability of our newly proposed Circuit Transformer Rewrite (ctrw) operator, we conduct experiments on 22 cases in IWLS 2023 contest benchmark that are within the computational capability of the computer we used to run ctrw in a reasonable time. The truth tables in the benchmarks are processed by the ABC command sequence read_truth -xf ex00.truth; collapse; sop; strash; dc2; suggested in [13] to generate the initial AIG circuits to be rewritten.

The following rewriting techniques are included as baselines:

- **Drw rewriting** [14]: the latest and best version of cut rewriting in ABC, which is of 4-inputs for each cut.
- **MFFW rewriting** [24]: a recent maximum fanout-free window rewriting with $k = 8$. MFFWs are optimized via resyn2 in ABC. Note that it is implemented in Python as the base of ctrw, which impacts its running efficiency.

For our proposed Circuit Transformer Rewriting, to evaluate the effectiveness of the iterative improvement training stage in

Algorithm 4, we report the results on two models, the initial model supervisedly trained on random circuits, and the enhanced model with iterative self-improvement training. To evaluate the additional flexibility of Circuit Transformer for rewriting that are proposed in Section 3.3 and Section 3.4, we report the result with and without these additional flexibility. For all the ctrw cases with MCTS, we set $M_{step} = 10$ and $M_{playout} = 10$.

The results are presented in Table 1. All of the AIGs generated by ctrw have successfully passed equivalence checking using cec in ABC, empirically demonstrating that the proposed neural generative method is capable of producing strictly feasible circuits.

We observe that ctrw with the enhanced model significantly outperforms the version with the initial model and also surpasses the MFFW baseline, which highlights the effectiveness of the proposed iterative self-improvement training. Despite the increased time cost, MCTS also shows improvement over direct generation. Furthermore, ctrw with guided DAG-aware rewriting markedly improves over the version without it, further showcasing its effectiveness.

5 CONCLUSION

In this work, we introduced the first logic synthesis operator powered by generative deep neural networks. This operator not only successfully generated strictly feasible circuits but also demonstrated significant effectiveness in reducing circuit size. The application of the Circuit Transformer model provides additional flexibility in rewriting, showcasing the synergy between traditional optimization techniques and modern deep learning models.

This paper paves the way for generative AI-based approaches in logic synthesis. Moving forward, our focus will be on scaling up the Circuit Transformer using large-scale AI platforms and enhancing both the efficiency and effectiveness of the model with state-of-the-art AI techniques.

REFERENCES

[1] Armin Biere. 2007. *The AIGER And-Inverter Graph (AIG) Format Version 20071012*. Technical Report 07/1. Institute for Formal Models and Verification, Johannes Kepler University, Altenbergerstr. 69, 4040 Linz, Austria.

[2] Jason Blocklove, Siddharth Garg, Ramesh Karri, and Hammond Pearce. 2023. Chip-Chat: Challenges and Opportunities in Conversational Hardware Design. In *2023 ACM/IEEE 5th Workshop on Machine Learning for CAD (MLCAD)*. 1–6.

[3] Robert Brayton and Alan Mishchenko. 2010. ABC: An Academic Industrial-Strength Verification Tool. In *Computer Aided Verification*. Springer Berlin Heidelberg, Berlin, Heidelberg, 24–40.

[4] Kaiyan Chang, Ying Wang, Haimeng Ren, Mengdi Wang, Shengwen Liang, Yinhe Han, Huawei Li, and Xiaowei Li. 2023. ChipGPT: How far are we from natural language hardware design. arXiv:2305.14019 [cs.AI]

[5] Antoine Grosnit, Cedric Malherbe, Rasul Tutunov, Xingchen Wan, Jun Wang, and Haitham Bou Ammar. 2022. BOiLS: Bayesian Optimisation for Logic Synthesis. In *2022 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. 1193–1196.

[6] Winston Haaswijk, Edo Collins, Benoit Seguin, Mathias Soeken, Frédéric Kaplan, Sabine Süsstrunk, and Giovanni De Micheli. 2018. Deep Learning for Logic Optimization Algorithms. In *2018 IEEE International Symposium on Circuits and Systems (ISCAS)*. 1–4.

[7] Abdelrahman Hosny, Soheil Hashemi, Mohamed Shalan, and Sherief Reda. 2020. DRiLLS: Deep Reinforcement Learning for Logic Synthesis. In *2020 25th Asia and South Pacific Design Automation Conference (ASP-DAC)*. 581–586.

[8] Guyue Huang, Jingbo Hu, Yifan He, Jialong Liu, Mingyuan Ma, Zhaoyang Shen, Juejian Wu, Yuanfan Xu, Hengrui Zhang, Kai Zhong, Xuefei Ning, Yuzhe Ma, Haoyu Yang, Bei Yu, Huazhong Yang, and Yu Wang. 2021. Machine Learning for Electronic Design Automation: A Survey. *ACM Trans. Des. Autom. Electron. Syst.* 26, 5, Article 40 (jun 2021), 46 pages.

[9] Donald E Knuth. 2020. *The Art of Computer Programming, Volume 4, Fascicle 5: Mathematical Preliminaries Redux; Introduction to Backtracking; Dancing Links*. Addison-Wesley Professional.

[10] Wenzhe Li, Hao Luo, Zichuan Lin, Chongjie Zhang, Zongqing Lu, and Deheng Ye. 2023. A Survey on Transformers in Reinforcement Learning. *Transactions on Machine Learning Research* (2023). https://openreview.net/forum?id=r30yuDPvf2 Survey Certification.

[11] Xihan Li, Xing Li, Lei Chen, Xing Zhang, Mingxuan Yuan, and Jun Wang. 2024. Circuit Transformer: End-to-end Circuit Design by Predicting the Next Gate. arXiv:2403.13838 [cs.LG]

[12] Mingjie Liu, Teo Ene, Robert Kirby, Chris Cheng, Nathaniel Pinckney, Rongjian Liang, Jonah Alben, Himyanshu Anand, Sanmitra Banerjee, Ismet Bayraktaroglu, et al. 2023. ChipNeMo: Domain-Adapted LLMs for Chip Design. *arXiv preprint arXiv:2311.00176* (2023).

[13] Alan Mishchenko and Satrajit Chatterjee. 2022. IWLS 2022 Programming Contest. https://www.iwls.org/iwls2022/contest/IWLS_2022_Programming_Contest.pdf

[14] A. Mishchenko, S. Chatterjee, and R. Brayton. 2006. DAG-aware AIG rewriting: a fresh look at combinational logic synthesis. In *2006 43rd ACM/IEEE Design Automation Conference*. 532–535. https://doi.org/10.1145/1146909.1147048 ISSN: 0738-100X.

[15] Walter Lau Neto, Max Austin, Scott Temple, Luca Amaru, Xifan Tang, and Pierre-Emmanuel Gaillardon. 2019. LSOracle: a Logic Synthesis Framework Driven by Artificial Intelligence: Invited Paper. In *2019 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. 1–6.

[16] OpenAI. 2022. Introducing ChatGPT. https://openai.com/blog/chatgpt

[17] Franz-Xaver Reichl, Friedrich Slivovsky, and Stefan Szeider. 2023. Circuit Minimization with QBF-Based Exact Synthesis. *Proceedings of the AAAI Conference on Artificial Intelligence* 37, 4 (June 2023), 4087–4094. https://doi.org/10.1609/aaai.v37i4.25524

[18] Heinz Riener, Winston Haaswijk, Alan Mishchenko, Giovanni De Micheli, and Mathias Soeken. 2019. On-the-fly and DAG-aware: Rewriting Boolean Networks with Exact Synthesis. In *2019 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, Florence, Italy, 1649–1654. https://doi.org/10.23919/DATE.2019.8715185

[19] David Silver, Aja Huang, Chris J. Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. 2016. Mastering the game of Go with deep neural networks and tree search. *Nature* 529, 7587 (Jan. 2016), 484–489. https://doi.org/10.1038/nature16961 03249.

[20] David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, et al. 2017. Mastering the game of go without human knowledge. *nature* 550, 7676 (2017), 354–359.

[21] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Ł ukasz Kaiser, and Illia Polosukhin. 2017. Attention is All you Need. In *Advances in Neural Information Processing Systems*, Vol. 30. Curran Associates, Inc.

[22] Hongkun Yu, Chen Chen, Xianzhi Du, Yeqing Li, Abdullah Rashwan, Le Hou, Pengchong Jin, Fan Yang, Frederick Liu, Jaeyoun Kim, and Jing Li. 2020. Tensor-Flow Model Garden. https://github.com/tensorflow/models.

[23] Keren Zhu, Mingjie Liu, Hao Chen, Zheng Zhao, and David Z. Pan. 2020. Exploring Logic Optimizations with Reinforcement Learning and Graph Convolutional Network. In *2020 ACM/IEEE 2nd Workshop on Machine Learning for CAD (MLCAD)*. 145–150.

[24] Xuliang Zhu, Ruofei Tang, Lei Chen, Xing Li, Xin Huang, Mingxuan Yuan, Weihua Sheng, and Jianliang Xu. 2023. A Database Dependent Framework for K-Input Maximum Fanout-Free Window Rewriting. In *2023 60th ACM/IEEE Design Automation Conference (DAC)*. 1–6. https://doi.org/10.1109/DAC56929.2023.10247727