ORL-LO: Offline Reinforcement Learning for Pretraining and Finetuning in Logic Optimization

GUANGDE DONG*, Beijing University of Posts and Telecommunications, China
HONGTAO CHENG*, Beijing University of Posts and Telecommunications, China
JIANWANG ZHAI†, Beijing University of Posts and Telecommunications, China
XIAO YANG, Beijing University of Posts and Telecommunications, China
CHUAN SHI, Beijing University of Posts and Telecommunications, China
KANG ZHAO, Beijing University of Posts and Telecommunications, China

As a key step in digital integrated circuit (IC) design, logic synthesis involves various logic optimization algorithms, where the quality of results (QoR) depends heavily on the optimization sequence used. Exploring the optimization space is challenging as the number of potential optimal permutations grows exponentially. Traditional methods rely on manual adjustments by experts, but are difficult to deal with complex and different circuits, leading to significant optimality gaps. Many automatic methods have been introduced, but they still face problems of low generalization and low efficiency.

In this work, we propose ORL-LO, a two-stage learning framework that leverages an offline reinforcement learning (RL) method (i.e., advantage-weighted actor-critic, AWAC) to pretrain on expert trajectories and finetune on target test circuits, enabling efficient exploration of optimal synthesis flows. Firstly, ORL-LO uses AWAC to pretrain a fast and high-performance intelligent policy to fully leverage the offline knowledge of a large corpus of high-quality expert trajectories. Then, the pretrained policy is finetuned for target circuits using AWAC and policy distillation to obtain better results. Compared with the state-of-the-art (SOTA) method, our framework can effectively improve the quality of logic optimization and significantly speed up the exploration time.

1 INTRODUCTION

Logic synthesis converts a high-level circuit description at the register transfer level (RTL) into an optimized gate-level netlist, involving multiple stages, including translation, technology-independent optimization, and technology mapping. The goal of technology-independent optimization is to apply a series of optimization algorithms that reduce the depth and number of nodes in the circuit's Boolean network, thereby enhancing the circuit's performance and quality. ABC [1] is a popular open-source logic synthesis tool that utilizes an and-inverter graph (AIG) as its subject graph and

---

[*] Co-first authors with equal contribution.

[†] Corresponding author.

Authors' addresses: Guangde Dong, School of Integrated Circuits, Beijing University of Posts and Telecommunications, Beijing, China, dongguande@bupt.edu.cn; Hongtao Cheng, School of Computer Science, Beijing University of Posts and Telecommunications, Beijing, China, chenghongtao@bupt.edu.cn; Jianwang Zhai, School of Integrated Circuits, Beijing University of Posts and Telecommunications, Beijing, China, zhaijw@bupt.edu.cn; Xiao Yang, School of Integrated Circuits, Beijing University of Posts and Telecommunications, Beijing, China, yangxiao@bupt.edu.cn; Chuan Shi, School of Integrated Circuits, Beijing University of Posts and Telecommunications, Beijing, China, shichuan@bupt.edu.cn; Kang Zhao, School of Computer Science, Beijing University of Posts and Telecommunications, Beijing, China, zhaokang@bupt.edu.cn.

1084-4309/2026/1-ART $15.00
https://doi.org/10.1145/3774323

provides heuristic synthesis flows like *resyn2* and *compress2* to optimize the structure of AIGs. Although this synthesis workflow is widely adopted, it can sometimes result in suboptimal results due to the different optimization strategies required by various circuits.

As the demand for higher QoR continues to increase, machine learning (ML) techniques are used to classify or predict the final QoR of these synthesis flows, as documented in studies [2, 3], have been proposed. However, these prediction methods suffer from limited accuracy and are difficult to explore different circuits efficiently. Furthermore, people use reinforcement learning (RL) to conceptualize the generation of synthesis flows as Markov decision processes (MDPs) to generate synthesis sequences autonomously. The DRiLLS framework [4] introduces an advanced system enabling an A2C agent to select optimal transformations flexibly. Zhu et al. [5] employ graph neural networks (GNNs) to capture the topology of AIGs and integrate it with historical decisions, thereby enriching the state information to enhance decision-making efficacy. Zhou et al. [6] apply random forests to conduct feature importance analysis, enabling feature pruning and investigating the generalization capabilities of RL within this context.

Additionally, researchers have begun to explore customized synthesis flows tailored to specific circuits to further enhance the quality of logic synthesis. Online learning frameworks like CBTune [7] and AlphaSyn [8] represent advanced strategies in logic synthesis. CBTune [7] employs a contextual bandit algorithm with the Syn-LinUCB algorithm, navigating the solution space efficiently and avoiding local optima by considering circuit characteristics and long-term payoffs. AlphaSyn [8] utilizes a domain-specific Monte Carlo tree search (MCTS) approach with a SynUCT algorithm for meticulous selection and a parallel exploration process, which has demonstrated superiority over traditional methods in terms of area reduction and runtime.

Online learning methods can provide customized optimization, but face challenges in generalization across different scenarios. Most existing methods require fresh learning and exploration for each new circuit, lacking effective utilization of existing data and knowledge. Just like humans, we can learn offline knowledge from a large amount of existing data and use it to handle and improve new situations. Therefore, an ideal policy should consider how to learn general offline knowledge from a vast array of existing circuits and synthesis flows and finetune it for unfamiliar circuits. Previous studies (e.g., RL4LS [6]) have demonstrated some generalization capabilities of RL; however, these findings are preliminary and necessitate further experimentation and discussion. Our preliminary work, PIRLLS [9], proposes using imitation learning to pretrain a policy network and leverage offline knowledge. However, imitation learning is inherently limited to mimicking expert behavior, restricting its ability to surpass expert performance. In recent years, offline RL methods [10–12] have been introduced, providing new approaches for extracting and utilizing offline knowledge. Unlike imitation learning, these methods enable policies to learn beyond expert demonstrations, allowing them to achieve superior performance by effectively leveraging large-scale offline data without the need for frequent retraining.

As illustrated in Fig. 1, ORL-LO first utilizes a heuristic algorithm to generate a large number of expert trajectories, uses the AWAC algorithm to pretrain policies, and finally uses the AWAC algorithm and policy distillation to finetune these pretrained policies on specific circuits. The framework facilitates faster exploration of target circuits by effectively leveraging offline knowledge. Our contributions can be summarized as follows:

(1) To obtain high-quality offline knowledge, we design a genetic algorithm (GA) to efficiently generate a large number of expert trajectories for training circuits.
(2) To better characterize the circuit state, we use carefully designed scalar features and a pretrained BERT model to extract features from AIG and historical operations.

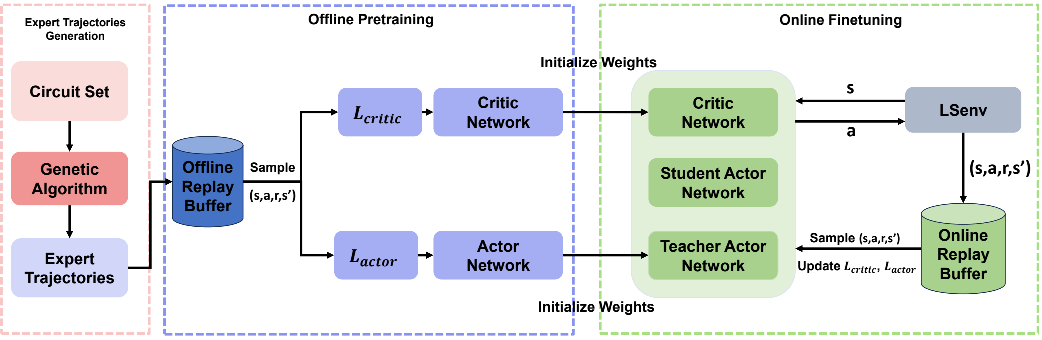


Fig. 1. Illustration of ORL-LO framework.

(3) To better capture the structural and functional properties of AIGs, we design a QoR prediction task to pretrain the GNN-based model, thus improving its ability to guide optimization decisions.
(4) We use the AWAC algorithm to learn offline knowledge from expert trajectories, and the pretrained policy can optimize unseen circuits rapidly and effectively.
(5) For more targeted optimization of target circuits, we use AWAC and policy distillation to finetune the pretrained policy and maintain its advantage of efficient exploration.
(6) We adopt a two-stage operator selection strategy: the policy is first pretrained within a fixed operator set, and then is allowed to be finetuned in a broad and variable operator space, enabling more flexible exploration. This improves search efficiency, reduces redundancy, and enhances generalization.

2 PRELIMINARIES

2.1 Logic Optimization and Technology Mapping

In digital IC design, logic optimization and technology mapping are critical steps for enhancing circuit performance. Logic optimization improves a circuit's efficiency and compactness by transforming and simplifying its logical representation—commonly in the form of an AIG. This process reduces power consumption, area, and delay without altering the circuit's functionality. Several key transformation operators provided by ABC [1] are commonly used:

- **Rewriting** [13]: A fast and greedy local optimization algorithm that replaces small AIG subgraphs with smaller pre-computed alternatives to reduce area. It operates iteratively on selected cuts and preserves functionality while minimizing node count.
- **Refactoring**: An extension of rewriting that applies to larger cuts (typically 10–20 inputs). It uses Binary Decision Diagrams (BDDs) and Sum-of-Products (SOP) representations to restructure deeper subgraphs, allowing the recovery of further redundancy beyond what rewriting alone can achieve.
- **Balancing**: A linear-time technique that restructures the AIG using associative transforms such as $a(bc) = (ab)c = (ac)b$, aiming to reduce circuit depth (logic levels). It is often interleaved with area-focused methods like rewriting to simultaneously achieve low delay and area.
- **Resubstitution** [14]: Attempts to re-implement the function of a node using other nodes (called divisors) already present in the network. If a more efficient expression is found, the node's structure is updated. This technique is particularly effective in removing redundant logic and enhancing sharing.

Technology mapping translates the optimized logical network into a physical gate-level netlist using a specific technology library. This step ensures that the circuit design is manufacturable with the chosen technology, taking into account the available gate types and their characteristics.

The results of logic optimization directly influence the effectiveness of technology mapping, determining the area and delay characteristics of the final netlist. Frameworks like ABC [1] systematically provide many transformation strategies to optimize the QoR of the circuit. The focus of this paper is to optimize the number of 6-LUTs in the FPGA process by utilizing various operators provided by ABC [1]. This involves exploring the design space through the combination and permutation of multiple operators, allowing for a comprehensive optimization strategy.

2.2 Reinforcement Learning

RL is a key area of machine learning, involving states, actions, and rewards. A state represents the current condition of the environment, serving as the basis for decision-making. The agent selects an action based on its policy, which is a mapping from states to actions. The chosen action interacts with the environment, leading to a new state and a corresponding reward signal. The reward reflects the immediate feedback from the environment and contributes to the cumulative reward that the agent aims to maximize over time. The policy serves as a guide for the agent's decision-making, evolving continuously through training to optimize its performance.

Let $\tau = (s_t, a_t, r_t)$ represent a sequence of state, action, and reward tuples. The goal of RL is to find a policy $\pi_\theta(a \mid s)$ that maximizes the expected sum of discounted future rewards. At each time step t, the state s_t describes the environment's current conditions and serves as the input to the policy. The action a_t, chosen based on $\pi_\theta(a \mid s_t)$, affects the environment and leads to the next state. The reward r_t is received after taking action a_t and indicates how effective the action was. The policy $\pi_\theta(a \mid s)$ aims to select the optimal action a given the state s, with its parameters θ being adjusted through learning to improve the policy's performance. The optimal policy π^* is defined as:

$$\pi^* = \arg\max_{\pi} \mathbb{E}_{\tau\sim\pi}[R_T], \quad \text{where } R_T = \sum_{t=1}^{T} \gamma^{t-1} r_t, \tag{1}$$

where γ is discount factor used to control future rewards, and T represents the length of the trajectory τ.

Offline RL, also known as batch RL, is a paradigm where the agent learns policies exclusively from a fixed dataset $\mathcal{D} = \{\tau_j\}_{j=1}^{N}$ of precollected trajectories, without interacting with the environment during training. This contrasts with online RL, which requires costly environment interactions for data collection. Compared with imitation learning, offline RL demonstrates superior capability in not only acquiring expert behaviors but also enhancing policy performance via objective function optimization, whereas imitation learning merely replicates expert demonstrations without optimization potential. The key challenge in offline RL lies in avoiding *distributional shift* [15]—the discrepancy between the state-action distribution in $\mathcal{D}$ and those induced by the learned policy. To address this, methods often incorporate **conservative policy updates** or **behavior regularization** to constrain the policy near the dataset's support. Recent studies emphasize the practical importance of detecting and mitigating such distributional shifts in real-world applications [16, 17]. For instance, the GradNorm approach [16] introduces a gradient-based method to detect out-of-distribution (OOD) inputs, showing that gradient norms vary meaningfully between in- and out-of-distribution data. The WILDS benchmark [17] further highlights that standard ML models suffer significant performance drops under realistic distribution shifts, motivating the need for robust offline RL approaches.

While constrained offline RL algorithms [10–12] perform well in offline settings, they often struggle to improve during finetuning. The core issue lies in accurately modeling behavior as new online data is collected. A behavior model constrains policy optimization to prevent distributional shift and mitigate extrapolation errors. In offline training, it is trained once via maximum likelihood estimation to capture the original data distribution. However, online finetuning must be continuously updated to incorporate new data, which is challenging due to the complex, multi-modal nature of the mixed offline and online distributions. As training progresses, behavior model accuracy deteriorates, leading to overly conservative constrained optimization, limiting policy improvement, and hindering finetuning effectiveness.

The advantage-weighted actor-critic (AWAC) [18] algorithm, an offline RL method, addresses distributional shifts through behavior regularization. Unlike methods that rely on explicit behavior models, AWAC employs an implicit constraint by weighting policy updates with advantage estimates, avoiding the need for direct behavior modeling. This approach mitigates the challenges associated with fitting an accurate behavior model during online finetuning, where the mixed distribution of offline and online data can lead to degraded model accuracy and overly conservative policy updates. By circumventing the reliance on behavior models, AWAC allows for more flexible adaptation during finetuning, leading to improved policy performance. We will provide a more detailed exposition on AWAC in subsequent sections. In our framework, the AWAC-based offline RL approach leverages offline knowledge for pretraining, thereby achieving efficient logic optimization.

2.3 Policy Distillation

In practice, learning optimal policies often faces challenges of complexity and high computational costs. To address this, we employ policy distillation [19], a technique that transfers knowledge from a complex teacher policy model to a lightweight student model. The core idea is to minimize the divergence between the output distributions of the two policies, enabling the student policy to mimic the teacher's behavior. Specifically, the Kullback-Leibler (KL) [20] divergence is usually used as the loss function, defined as:

$$L_{\mathrm{KL}} = \sum_{a \in A} \pi^{i}(a|s) \log\left(\frac{\pi^{i}(a|s)}{\pi^{0}(a|s)}\right), \tag{2}$$

where $\pi^{i}(a|s)$ represents the probability of the student policy selecting action a in the state s during the current iteration, and $\pi^{0}(a|s)$ denotes the corresponding probability for the teacher policy under the same state. By optimizing this loss function, the student policy progressively aligns its probability distribution with the teacher's, achieving efficient knowledge transfer. In this work, we leverage policy distillation to extract offline knowledge from pretrained policies and enhance the analysis of target circuits through targeted exploration.

2.4 Graph Neural Networks

GNNs are specialized deep learning architectures designed to process graph-structured data, where information is represented as nodes and their connections as edges. GNNs operate through message passing mechanisms, where each node aggregates information from its neighbors to update its own representation. This process enables GNNs to capture both node features and topological structures simultaneously. Formally, for a node v with features $\mathbf{h}_v^{(l)}$ at layer l, the update rule can be represented as:

$$\mathbf{h}_v^{(l+1)} = \mathrm{UPDATE}(\mathbf{h}_v^{(l)}, \mathrm{AGGREGATE}(\{\mathbf{h}_u^{(l)} : u \in \mathcal{N}(v)\})), \tag{3}$$

where $\mathcal{N}(v)$ denotes the neighborhood of node v.

In the context of logic circuits, GNNs naturally model netlists with logic gates as nodes and connections as edges, allowing them to learn from circuit topology. This capability makes GNNs particularly valuable for tasks like logic optimization and technology mapping, as they can identify structural patterns that influence circuit performance and guide optimization decisions based on both local component properties and global circuit structure [21]. Recent advances have further extended the applicability of GNNs by designing architectures that incorporate heterogeneous information sources from the chip design flow. For example, circuit representation learning [22–26] has demonstrated the potential of GNNs across multiple EDA tasks. These developments illustrate the growing role of GNNs as general-purpose models in electronic design automation.

2.5 BERT

BERT [27] is a pretrained deep learning model based on the Transformer architecture. The model is trained on large amounts of text data and learns deep language features through two tasks: masked language model (MLM) and next sentence prediction (NSP). One of the core features of BERT is its bidirectional representation capability, which allows it to consider both the left and right context of each word simultaneously, producing richer and more accurate word embeddings. This makes BERT excel in various natural language processing tasks such as text classification, question answering, and language understanding. In this work, we use BERT to convert variable-length historical operator information into fixed-length vectors, helping the agent to better understand and utilize past operations, thus enhancing its decision-making capabilities.

3 PROBLEM DEFINITION AND MOTIVATION

3.1 Problem Definition

Various logic optimization algorithms, including balancing, rewriting, and refactoring, are used to optimize digital circuits. The arrangement and combination of different operators significantly affect the final optimization outcome, necessitating exploration within a vast design space. Defining $A = \{n_1, n_2, \ldots, n_m\}$ as the set of available optimizations in a logic synthesis tool and letting k be the length of optimized synthesis flows, there exist m^k possible flows, creating an exponentially large search space. This makes finding the optimal flow for complex circuit designs particularly challenging. In this paper, the objective is to identify the optimal flow for the target circuit, i.e., the best combination of optimization operators for a given length.

3.2 Motivation

3.2.1 Generalization of Operator Sequences

Although optimal synthesis flows for different circuits may vary, useful knowledge can still be extracted from existing flows to aid new designs. For circuit designs that have certain similarities, their optimal flows are often also similar. We verify this experimentally, as shown in Table 1. GA (Algorithm 1 in Section 4.2) is employed to find well-performing synthesis flows for similar circuits. Firstly, we select several pairs of similar circuits, such as S838/S838.1 and C7552/c7552, based on the cosine similarity of embeddings generated by the pretrained GNN introduced in Section 4.3. For each circuit, we preserve the top-10 synthesis flows discovered by GA exploration and report their average 6-LUT counts and standard deviations in the "Best" column. Then, we swap the best flows between similar circuits and evaluate them on the target circuit, with the average and standard deviation of the results recorded in the "Change" column. As a baseline, we also consider resyn2, a fixed synthesis script provided by ABC [1]. Although not necessarily the state of the art, resyn2 is widely accepted as a reproducible reference for evaluating optimization performance. In our experiments, resyn2 was executed twice, and the results are reported in the "resyn2*2" column,

Table 1. The average number of 6-LUTs for structurally similar circuits over ten synthesis flows

Circuit	Best ($\mu \pm \sigma$)	Change ($\mu \pm \sigma$)	resyn2*2
s838	60.00 ± 0.00	63.20 ± 1.23	65 ± 0.00
s838.1	61.00 ± 0.00	67.20 ± 1.87	70 ± 0.00
C7552	344.80 ± 0.42	369.40 ± 1.17	382 ± 0.00
c7552	333.00 ± 0.00	364.20 ± 1.03	382 ± 0.00
C3540	218.00 ± 2.11	212.40 ± 0.52	230 ± 0.00
c3540	212.40 ± 0.52	218.00 ± 2.11	230 ± 0.00
C2670	110.00 ± 0.00	126.00 ± 0.00	131 ± 0.00
c2670	126.00 ± 0.00	110.00 ± 0.00	131 ± 0.00
Average	183.15 ± 1.01	191.30 ± 0.99	202.62 ± 0.00

with zero standard deviation since both runs yield identical outcomes. The results demonstrate that swapping flows between similar circuits consistently outperforms resyn2*2. This finding suggests that exploiting historical synthesis data from existing designs can substantially improve the optimization of new circuit designs.

Based on the above analysis, we propose a two-phase method that combines offline policy training with online finetuning to optimize the synthesis flow of specific AIG. In the offline training phase, we aim to leverage offline RL to obtain a strategy that mimics expert trajectories and can be used to optimize AIGs with similar structures. While the offline strategy demonstrates a certain level of generalization to AIGs of similar structures, the policy still needs to be finetuned for specific designs to achieve optimal optimization results. This tailored approach ensures that each design benefits from both broad foundational insights and specific, targeted adjustments, maximizing the effectiveness of the synthesis flow.

3.2.2 Off-Policy RL for Sample-Efficient Optimization

In the previous section, we observed that similar circuits may benefit from similar operator sequences. So, how should we acquire offline knowledge? One of the simplest ways to utilize prior data, such as demonstrations for RL, is to pretrain a policy with imitation learning, and finetune with on-policy RL [9]. This approach has two drawbacks: (1) prior data may not be optimal; (2) on-policy finetuning is data inefficient as it does not reuse the prior data in the RL stage. The computational bottleneck in ABC optimization operators motivates our adoption of off-policy RL, where environment interactions account for over 90% of total runtime. Traditional on-policy methods exhibit suboptimal sample efficiency due to their dependence on fresh environment interactions for each policy update. Our framework addresses this challenge through three coordinated mechanisms: (1) Experience replay strategically reuses historical synthesis trajectories stored in buffer $\mathcal{B} = \{\tau_j\}$, enabling multiple policy updates per environmental interaction cycle; (2) Delayed policy updates employ Q-function regularization to decouple policy improvement from immediate data collection; (3) Target network stabilization maintains independent policy π_θ and target networks π_{θ^-} to prevent training divergence.

4 ORL-LO FRAMEWORK

4.1 Overview of ORL-LO

As inferred in Section 3.2, circuits with similar structures may benefit from similar optimal synthesis flows. Based on this insight, we introduce ORL-LO, as illustrated in Fig. 1. This method employs a heuristic genetic algorithm to explore circuits in the training set, producing the optimal synthesis flows as expert trajectories. It then extracts circuit features from these trajectories and concatenates

Algorithm 1 Expert Trajectory Generation

```
1: Input: $G$: Number of generations, $P$: Population size, $C_p$: Crossover probability, $M_p$: Mutation probability, $C$: Stopping condition for unchanged fitness
2: Output: Optimal synthesis flow
3: Initialize population with random gene values   ▷ Randomly initialize the gene sequences representing synthesis flows.
4: for $g = 1$ to $G$ do
5:     Evaluate the fitness of each individual   ▷ Compute fitness for all flows.
6:     if fitness unchanged for $C$ consecutive generations then
7:         Break   ▷ Early stopping if no improvement is observed.
8:     end if
9:     Select parents via tournament selection   ▷ Choose high-fitness flows as parents using tournament selection.
10:    for $i = 1$ to $P$ do
11:        Crossover selected parents with probability $C_p$   ▷ Apply uniform crossover to generate offspring.
12:        Add offspring to new population   ▷ Store newly generated flows in the next generation.
13:    end for
14:    for $i = 1$ to $P$ do
15:        Mutate genes based on $M_p$   ▷ Apply random mutations to introduce diversity.
16:    end for
17: end for
```

them into numerous (s_t, a_t) pairs, representing the "expert's" choice of optimization operator a_t at circuit state s_t. Importantly, the offline RL method AWAC is used to pretrain the policy network based on these state-action pairs. This policy network can efficiently optimize new circuits with similar features. However, due to the differences between circuits, the pretrained policy may not achieve the best results on all target designs. Therefore, we proceed with finetuning using AWAC and policy distillation. Compared to imitation learning, offline RL methods can learn better policies through objective function optimization, thereby surpassing the behavior demonstrated by experts and improving performance. In summary, ORL-LO consists of three main components: generating expert trajectories through a genetic algorithm, pretraining in a fixed operator space using AWAC, and finetuning in a designated operator space based on circuit features using AWAC and policy distillation.

4.2 Trajectory Generation

To learn useful prior knowledge from existing designs, we need high-quality synthesis flows for each design. Therefore, to obtain the dataset for pretraining, instead of using existing synthesis flows, we design a fast genetic algorithm (GA) to generate a large number of high-quality expert trajectories. To achieve better logic optimization effects and greater optimization possibilities, we have selected 7 operators, including "rewrite", "rewrite -z", "balance", "refactor -z", "refactor", "resub", "resub -z". The above operators are widely used in logic synthesis tasks to ensure generality.

The detailed algorithm is shown in Algorithm 1. The genetic algorithm abstracts candidate solutions as genes, defined by various operators, and measures fitness as the negative of the 6-LUT count. In the initialization stage (Lines 1–3), candidate synthesis flows are represented as

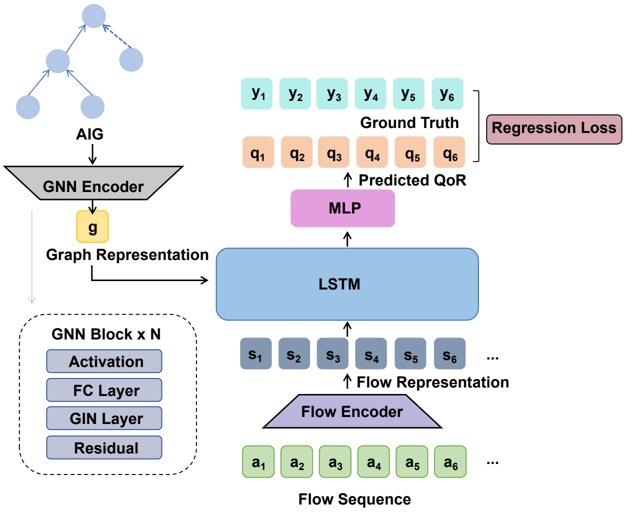


Fig. 2. The structure of GNN-based QoR predictor.

genes, randomly initialized, and their fitness is evaluated based on the resulting number of 6-LUTs. An early stopping criterion (Lines 6–8) ensures termination when no improvement in fitness is observed across consecutive generations. Tournament selection (Line 9) follows, where a subset of individuals is selected randomly, and the one with the highest fitness becomes a parent for the next generation. This repeats until sufficient parents are chosen. Uniform crossover (Lines 10-12) then allows parents to exchange genes at each position with a certain probability, creating new offspring and restoring the population to its original size. The mutation (Lines 14-15) is applied to introduce diversity. Through iterations, less fit individuals are eliminated, enhancing the results significantly. Multiprocessing improves the efficiency of fitness calculations. The optimal synthesis flow obtained for each design is determined as the "expert trajectory" and constitutes the pretraining dataset.

4.3 Pretraining of QoR-aware GNN

Circuits are often abstracted as graph-structured data, where scalar features extracted from circuits alone fail to comprehensively capture the complex structural information inherent in these designs. GNNs, as deep learning models specifically designed for graph-structured data, can effectively capture these structural characteristics through message passing between nodes, making them particularly suitable for circuit optimization tasks. Inspired by previous work [28, 29], we use QoR prediction as the pretraining objective for our GNN-based model to fully leverage the offline trajectory data. QoR prediction aims to forecast the number of 6-LUTs required after applying an operator sequence to the circuit graph and subsequent FPGA mapping. By systematically training GNN weights on extensive offline datasets, GNNs can learn the intrinsic relationships between circuit structures and their performance characteristics. This pretraining strategy not only enhances the training efficiency of subsequent online exploration but also improves the final results of logic optimization.

In the pretraining process, we employ the number of 6-LUTs as the primary metric for evaluating circuit QoR, and develop a hybrid predictive model to estimate optimization potential. For each circuit C_i in our dataset, we capture expert-derived optimization trajectories τ^* that represent optimal performance paths. Each trajectory τ^* comprises a sequence of optimization operations demonstrating superior circuit improvement strategies.

Methodologically, we employ two complementary neural network architectures to encode circuit features (as shown in Fig. 2): (1) Long Short-Term Memory (LSTM) networks capture the sequential characteristics of optimization trajectories, generating a sequential representation $\mathbf{h}_s$; (2) GNN encodes the structural properties of AIGs, producing a structural representation $\mathbf{h}_g$. Then, we concatenate these representations and employ a multi-layer perceptron (MLP) to predict post-optimization circuit performance:

$$\hat{y} = \mathrm{MLP}\left([\mathbf{h}_s; \mathbf{h}_g]\right). \tag{4}$$

Recognizing the pronounced disparity in circuit scales (tens to tens of thousands of 6-LUTs), we introduce a normalization strategy that improves generalization across designs. Instead of predicting absolute 6-LUT quantities, we target the QoR ratio $r_i = \frac{L_{\mathrm{opt},i}}{L_{\mathrm{init},i}}$, where $L_{\mathrm{opt},i}$ represents the optimized 6-LUT count for circuit C_i, and $L_{\mathrm{init},i}$ denotes the initial 6-LUT count before optimization.

The loss function for QoR-aware GNN pretraining utilizes mean squared error (MSE) to quantify prediction accuracy across N circuit samples between ground truth r_i and predicted ratio $\hat{r}_i$:

$$\mathcal{L} = \frac{1}{N}\sum_{i=1}^{N}(r_i - \hat{r}_i)^2. \tag{5}$$

Furthermore, we exclude small-scale circuits from the training set, as these circuits have relatively limited optimization space, simple structures, and uniform variation patterns, which could cause the model to overfit small-scale circuit features.

4.4 RL Environment

State Representation. We divide the state in the environment into three parts: a set of metrics that depict the current circuit design, the raw AIGs, and the historical actions.

First, as outlined in Section 2.3, characterizing the state of the AIG circuit for optimization is crucial. A feature extraction algorithm for AIGs was developed to better represent the circuit state for optimization, inspired by the feature importance analysis in RL4LS [6]. We identified 14 key scalar features, listed in Table 2, including input/output count, gate count, levels, width (maximum nodes per layer), mapped 6-LUTs count and levels, average and standard deviation of fan-out for input nodes and all AND gates, ratio of AND to NOT gates, and AND gate reduction ratio. These features are normalized to improve training and inference efficacy.

Table 2. Scalar features of AIG circuits

ID	Feature	ID	Feature
1	Input	2	Output
3	Number of gates	4	Level
5	Width	6	Number of LUTs
7	Number of LUTs level	8	Avg of input node fan-out
9	Std of input node fan-out	10	Avg of and node fan-out
11	Std of and node fan-out	12	And_nodes percent
13	Not_nodes percent	14	And_node_reduction_percent

Furthermore, the historical operations optimized for AIGs can also provide some state features. To better utilize this historical operation information, we use a pretrained BERT [27] model to capture these features. The reason for choosing BERT is that the length of historical operations is variable, and BERT can convert this variable length into a fixed-length vector, thus more effectively capturing the contextual relationships within the optimization flow. Ablation studies have demonstrated the advantages of this approach.

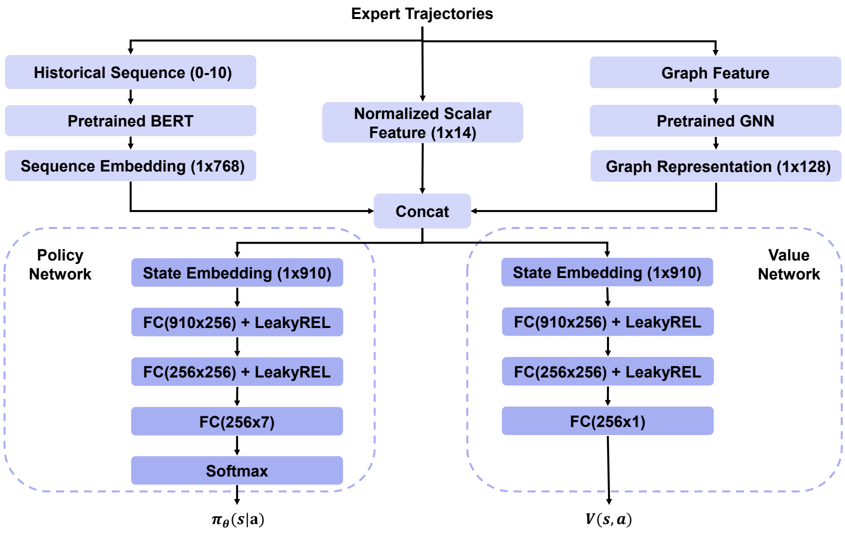


Fig. 3. Feature extraction and policy network architecture.

Finally, for understanding the structure and relationships within the AIG, graph features are crucial, and a pretrained GNN is used to extract and process these features. The GNN efficiently captures the structural information of the graph and outputs a graph representation, which is a fixed-length vector. This representation provides important contextual information for the policy network, enabling it to make more informed optimization decisions.

These features are then concatenated into a unified 910-dimensional state representation, as shown in Fig. 3, which serves as the input to the policy and value networks. Both networks consist of two fully connected (FC) layers with LeakyReLU activations, followed by a task-specific output layer. The policy network outputs a probability distribution $\pi_\theta(s, a)$ over candidate operators through a Softmax layer, while the value network predicts the expected reward $V(s, a)$ of the current state. The output dimension of the policy head (e.g., 256×7) corresponds to the number of available optimization operators and can be flexibly adjusted depending on the selected operator set.

Reward. Our RL environment iteratively applies ABC optimization commands to the initial circuit netlist. The state and action spaces align with imitation learning. After each action, the environment provides the next state and reward. Our goal is to find the optimal sequence to minimize the number of 6-LUTs. The reward function is:

$$r_t = \begin{cases} L_m - L(s_{t+1}), & \text{if } L(s_{t+1}) < L_m \\ -b, & \text{otherwise,} \end{cases} \quad L_{min} = \min(L_m, L(s_{t+1})), \tag{6}$$

where L_m records the best-found 6-LUT count up to time t and $L_m = L(s_0)$ initially. For each time step t, if $L(s_{t+1})$ is better than L_m, we give a positive reward corresponding to the improvement; otherwise, a small penalty $-b$ is applied. This reward function effectively encourages continuous improvement.

4.5 Pretraining with AWAC

In Section 4.2, we introduced the method for generating expert trajectories. What is more important is how to learn offline knowledge from existing expert trajectories and obtain a high-quality pretrained policy network, which is achieved through AWAC in this work. Unlike online RL, offline RL (i.e., AWAC) trains the network to optimize the policy by learning from the expert trajectories, effectively improving the policy's performance through value-based objective functions. We are the first to apply the offline RL method AWAC to pretrain synthesis policies for cross-circuit generalization. This extends the applicability of offline RL beyond single-task settings to logic synthesis, where pretrained policies can be reused across structurally similar circuits.

The AWAC algorithm trains an off-policy critic and an actor with an implicit policy constraint. AWAC follows the design for actor-critic algorithms with a policy evaluation step to learn Q_π and a policy improvement step to update π. AWAC uses off-policy temporal-difference learning to estimate Q_π in the policy evaluation step, and a policy improvement update that is able to obtain the benefits of offline RL algorithms at training from prior datasets. We describe the policy improvement step in AWAC below, and then summarize the entire algorithm.

Policy improvement for AWAC proceeds by learning a policy that maximizes the value of the critic learned in the policy evaluation step via TD bootstrapping. If done naively, this can lead to the issues described in Section 2.3, but we can avoid the challenges of bootstrap error accumulation by restricting the policy distribution to stay close to the data observed thus far during the actor update, while maximizing the value of the critic. At iteration k, AWAC, therefore, optimizes the policy to maximize the estimated Q-function $Q^{\pi_k}(s,a)$ at every state s, while constraining it to stay close to the actions observed in the data, similar to prior offline RL methods, though this constraint will be enforced differently. Note from the definition of the advantage that optimizing $Q^{\pi_k}(s,a)$ is equivalent to optimizing $A^{\pi_k}(s,a)$. We can therefore write this optimization as:

$$\pi_{k+1} = \arg\max_{\pi\in\Pi} \mathbb{E}_{a\sim\pi(\cdot|s)}[A^{\pi_k}(s,a)], \tag{7}$$

$$\text{s.t. } D_{KL}(\pi(\cdot|s)\|\pi_\beta(\cdot|s)) \leq \epsilon. \tag{8}$$

As we saw in Section 2.3, enforcing the constraint by incorporating an explicit learned behavior model [10–12] leads to poor finetuning performance. Instead, we enforce the constraint implicitly, without learning a behavior model. We first derive the solution to the constrained optimization in Equation (7) to obtain a non-parametric closed form for the actor. This solution can be projected onto the parametric policy class without any explicit behavior model. The analytic solution to Equation (7) can be obtained by enforcing the Karush-Kuhn-Tucker (KKT) conditions. The Lagrangian is:

$$\mathcal{L}(\pi,\lambda) = \mathbb{E}_{a\sim\pi(\cdot|s)}[A^{\pi_k}(s,a)] + \lambda(\epsilon - D_{KL}(\pi(\cdot|s)\|\pi_\beta(\cdot|s))), \tag{9}$$

and the closed form solution to this problem is $\pi^*(a|s) \propto \pi_\beta(a|s)\exp\left(\frac{1}{\lambda}A^{\pi_k}(s,a)\right)$. When using function approximators, such as deep neural networks, as we do, we need to project the non-parametric solution into our policy space. For a policy π_θ with parameters θ, this can be done by minimizing the KL divergence D_{KL} of π_θ from the optimal non-parametric solution π^* under the data distribution $\rho_{\pi_\beta}(s)$:

$$\arg\min_\theta \ \mathbb{E}_{\rho_{\pi_\beta}(s)}\left[D_{KL}(\pi^*(\cdot|s)\|\pi_\theta(\cdot|s))\right] \tag{10}$$

$$= \arg\min_\theta \ \mathbb{E}_{\rho_{\pi_\beta}(s)}\left[\mathbb{E}_{\pi^*(\cdot|s)}\left[-\log\pi_\theta(\cdot|s)\right]\right]. \tag{11}$$

Note that the parametric policy could be projected with either direction of the KL divergence. Choosing the reverse KL results in explicit penalty methods [12] that rely on evaluating the density of a learned behavior model. Instead, by using forward KL, we can compute the policy update by sampling directly from β:

$$\theta_{k+1} = \arg\max_\theta \ \mathbb{E}_{s,a\sim\beta}\left[\log\pi_\theta(a|s)\exp\left(\frac{1}{\lambda}A^{\pi_k}(s,a)\right)\right]. \tag{12}$$

This actor update amounts to weighted maximum likelihood (i.e., supervised learning), where the targets are obtained by re-weighting the state-action pairs observed in the current dataset

Algorithm 2 Advantage-Weighted Actor-Critic (AWAC)

```
1: Input: Offline dataset $\mathcal{D} = \{(s, a, s', r)\}_j$
2: Initialize replay buffer $\beta = \mathcal{D}$    ▷ Initialize replay buffer with the offline dataset.
3: Initialize policy $\pi_\theta$ and Q-function $Q_\phi$    ▷ Initialize actor (policy) and critic (Q-function) networks.
4: for iteration $i = 1, 2, \ldots$ do
5:     Sample batch $(s, a, s', r) \sim \beta$    ▷ Sample a mini-batch from the replay buffer.
6:     $y = r(s, a) + \gamma \mathbb{E}_{s', a' \sim \pi_k(a'|s')} [Q_{\phi_k}(s', a')]$    ▷ Calculate Q-values. $Q_{\phi_k}$ is the critic network and $a'$ is from policy $\pi_k$.
7:     $\phi \leftarrow \arg\min_\phi \mathbb{E}_{(s,a,y)\sim\text{batch}} [(Q_\phi(s, a) - y)^2]$    ▷ Update critic by minimizing the mean squared Bellman error.
8:     $\theta \leftarrow \arg\max_\theta \mathbb{E}_{s,a\sim\beta} \left[\log \pi_\theta(a|s) \exp\left(\frac{1}{\lambda} A^{\pi_k}(s, a)\right)\right]$    ▷ Update Weights using advantage $A^{\pi_k}(s, a)$.
9: end for
```

by the predicted advantages from the learned critic, *without* explicitly learning any parametric behavior model, simply sampling (s, a) from the replay buffer β.

The full AWAC algorithm for offline RL with online finetuning is summarized in Algorithm 2.

In the initialization stage (Lines 1–3), the offline dataset is first stored in a replay buffer, and both the policy (actor) and Q-function (critic) networks are initialized. In the policy evaluation step (Lines 4–7), a mini-batch is first sampled from the replay buffer (Lines 4–5). Then, target Q-values are computed using TD bootstrapping (Line 6). The critic network parameters are subsequently updated by minimizing the mean squared Bellman error (Line 7). In the policy improvement step (Line 8), the actor parameters are updated according to Equation (12), where the update corresponds to a weighted maximum likelihood objective based on the estimated advantages.

In a practical implementation, we can parameterize the actor and the critic by neural networks and perform SGD updates from Equation (12). Through this iterative process of alternating policy evaluation and improvement, AWAC effectively leverages offline data.

4.6 Finetuning with AWAC and Policy Distillation

After offline RL pretraining, we obtain a pretrained policy that can effectively optimize new circuits by learning from expert behavior. However, since different circuits have variations, targeted adjustments need to be made according to the target circuit. We propose a novel dual-policy exploration strategy that combines a pretrained policy with a randomly initialized policy to jointly explore and improve performance. These policies are integrated via policy distillation to balance exploitation of prior knowledge and continued exploration. This is, to the best of our knowledge, the first use of dual-policy distillation in logic synthesis optimization.

4.6.1 Dual Policy Distillation.

We initialize the actor's parameters with the pretrained policy weights π_{BC}^{θ}. Finetuning is challenging when the policy over-converges, skewing the actor's output towards a single action. To address this, we propose dual-policy exploration: the pretrained policy guides initial high-quality trajectories, while a randomly initialized policy enables broader exploration, balancing imitation and flexibility.

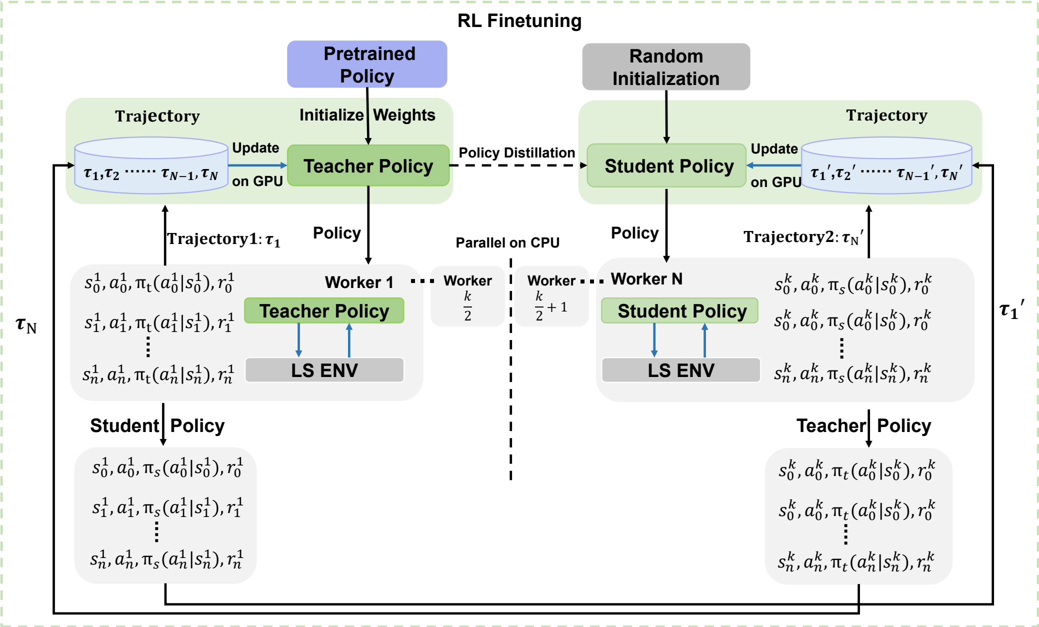

RL Finetuning
Pretrained Policy
Random Initialization
Trajectory
Initialize Weights
Trajectory
$\tau_1,\tau_2 \cdots\cdots \tau_{N-1},\tau_N$
Update
on GPU
Teacher Policy
Policy Distillation
Student Policy
Update
on GPU
$\tau_1',\tau_2' \cdots\cdots \tau_{N-1}',\tau_N'$
Trajectory1: τ_1
Policy
Parallel on CPU
Policy
Trajectory2: τ_N'
$s_0^1, a_0^1, \pi_t(a_0^1|s_0^1), r_0^1$
$s_1^1, a_1^1, \pi_t(a_1^1|s_1^1), r_1^1$
$s_n^1, a_n^1, \pi_t(a_n^1|s_n^1), r_n^1$
Worker 1
Teacher Policy
LS ENV
Worker $\frac{k}{2}$
Worker $\frac{k}{2}+1$
Worker N
Student Policy
LS ENV
$s_0^k, a_0^k, \pi_s(a_0^k|s_0^k), r_0^k$
$s_0^k, a_0^k, \pi_s(a_0^k|s_0^k), r_0^k$
$s_n^k, a_n^k, \pi_s(a_n^k|s_n^k), r_n^k$
τ_N
τ_1'
Student Policy
$s_0^1, a_0^1, \pi_s(a_0^1|s_0^1), r_0^1$
$s_1^1, a_1^1, \pi_s(a_1^1|s_1^1), r_1^1$
$s_n^1, a_n^1, \pi_s(a_n^1|s_n^1), r_n^1$
Teacher Policy
$s_0^k, a_0^k, \pi_t(a_0^k|s_0^k), r_0^k$
$s_0^k, a_0^k, \pi_t(a_0^k|s_0^k), r_0^k$
$s_n^k, a_n^k, \pi_t(a_n^k|s_n^k), r_n^k$

Fig. 4. Distributed training for RL finetuning methodology.

Due to the lengthy sample collection time, interaction with the environment becomes a training bottleneck. Therefore, we design a distributed training mechanism using multiple processes to improve sample collection efficiency, as shown in Fig. 4.

The RL finetuning framework employs two parallel policies: a pretrained teacher policy and a randomly initialized student policy. Both interact with separate subsets of logic synthesis environments to collect expert trajectories in parallel. Initially, the framework evenly distributes K environments between the pretrained teacher policy and the randomly initialized student policy, and gradually reduces the teacher's allocation over time. $\pi_s(a \mid s)$ denotes the student policy's action probability given state s, and $\pi_t(a \mid s)$ denotes the teacher's. "LSenv" denotes the logic synthesis environment for evaluating operator sequences and gathering optimization feedback. Each policy produces action sequences based on its own parameters. Importantly, both teacher and student not only update policies using their own data but also by cross-utilizing the other's trajectories to improve generalization and stability. This dual-policy design enables a synergy of exploitative guidance from the teacher and explorative learning from the student. The figure also highlights that all trajectory interactions occur in parallel across CPU workers, while model updates take place on the GPU.

To expedite the student actor's alignment with the teacher actor's performance, we utilize the principle of policy distillation and add the additional KL divergence loss L_{KL} (Equation (2)) to the standard AWAC's actor loss, i.e. L_{actor}.

This additional loss is intended to enable the student policy to rapidly achieve the performance levels of the teacher policy. Therefore, the student's loss is defined as:

$$L_{\text{student actor}} = L_{\text{actor}} + \alpha L_{\text{KL}}, \tag{13}$$

where α is a weighting factor. L_{actor} is mainly responsible for adjusting the policy to enhance the agent's performance in a specific environment by optimizing its behavior to maximize expected rewards.

To encourage exploration, we incorporate L_{Entropy} into $L_{\text{teacher actor}}$. L_{Entropy} is defined as:

$$L_{Entropy} = -\sum_{i=1}^{N} \sum_{s_t \in \tau^{(i)}} \sum_{a \in A} \pi(a|s_t) \cdot \log \pi(a|s_t), \tag{14}$$

And the teacher actor's loss is as follows:

$$L_{\text{teacher actor}} = L_{\text{actor}} + L_{\text{value}} + \beta L_{\text{Entropy}}, \tag{15}$$

where β is a weighting factor. By employing this approach, the pretrained policy can be finetuned using RL and policy distillation. This allows for better utilization of offline knowledge to optimize specific circuits without compromising efficiency.

4.6.2 *Selection of Operators.*

In the pretraining stage, we use 7 fundamental operators to pretrain the model, ensuring that it initially learns the basic patterns of logic optimization. Then, in the finetuning stage, we filter the entire operator space shown in Table 3 and retain only those operators that can improve area and level or have minimal negative impact on the area. Specifically, we select operators that lead to improvements in area or delay, or whose negative impact on these metrics is less than 10%. Based on this refined space, we perform RL finetuning.

Table 3. Evaluated optimization operators

ID	Operator	ID	Operator
1	rewrite	2	dc2
3	rewrite -z	4	ifraig
5	refactor -z	6	if -g
7	refactor	8	&sopb
9	balance	10	&blut
11	resub	12	&dsdb
13	resub -z	14	&b
15	multi -m; sop; fx	16	sweep; sop; fx

This approach offers several advantages. First, by learning general optimization patterns in a broad action space and then refining them within a more focused and effective operator set, the model achieves better generalization across different circuits. Second, reducing the operator space eliminates unnecessary exploration, significantly lowering computational overhead. Additionally, removing operators that negatively impact optimization quality enhances training efficiency and accelerates convergence.

To further improve adaptability, we leverage a pretrained model to initialize most of the network while redefining and randomly initializing the final layer for the specific optimization task. Specifically, we retain the pretrained feature extractor and replace the last fully connected layer with a new one that matches the output dimensions of our target task. This strategy allows us to benefit from the pretrained representations while enabling the model to learn task-specific decision boundaries effectively, ensuring a balance between optimization quality and computational efficiency.

4.6.3 *Dynamic Operator Length and Early Stopping Mechanism.*

During the optimization process, circuits often get stuck in local optima, where the area and level of the circuit are significantly worse compared to the initial circuit. In such cases, continuing the exploration is inefficient and unnecessary. Most existing research [4–8] typically fixes the length of the synthesis flow during the exploration, which can lead to resource wastage. Moreover, in RL, having more high-quality trajectories can improve the learning process.

Based on this analysis, we use a dynamic-length synthesis flow. Initially, the synthesis flow length is relatively short, allowing for quick iterations and early exploration. As the training progresses, the length of the synthesis flow adapts based on optimization progress, generally increasing over time to converge to the optimal length that is best suited for the specific circuit being optimized. Specifically, we apply a mechanism: if the area or delay does not improve for 8 consecutive steps, the current episode is terminated. This allows for efficient exploration, avoiding the unnecessary consumption of resources while ensuring deeper and more thorough optimization towards the end of the process.

Furthermore, to enhance the optimization process, we introduce an early stopping mechanism. The exploration process for a given circuit is terminated if more than 40 consecutive episodes fail to improve upon the current best area. This mechanism prevents unnecessary computation when further improvements are unlikely, thereby saving computational resources and time.

5 EVALUATION

Comprehensive experiments are conducted to evaluate the proposed ORL-LO framework. In Section 5.1, we first introduce the experimental settings. In Section 5.2, we give the optimization results and compare our method with previous work to prove its superiority. In Section 5.3, we analyze the impact of the number of operators on the optimization performance. In Section 5.4, we perform an ablation study on the ORL-LO framework.

5.1 Experiments Settings

The ORL-LO framework is implemented in Python, using PyTorch and the RL framework OpenAI Gym [30]. In addition, the exploration environment of ORL-LO is implemented by developing C++ interfaces for ABC [1]. The experiments are conducted on Intel(R) Xeon(R) 8383 CPU @ 2.60GHz with an NVIDIA RTX 3090 GPU. To enable efficient experience collection, we employ Python's multiprocessing module, launching 10 parallel processes, aligned with the configuration used in the CBTune baseline to ensure a fair comparison.

217 circuits from various well-known benchmark suites are used for pretraining, including ISCAS'85 [31], ISCAS'89 [32], ITC'99 [33], LGSynth'89 [34], LGSynth'91 [35], IWLS'93 [36], IWLS 2005 [37], and LEKO/LEKU benchmarks [38]. These benchmarks provide diverse circuits that guarantee the performance and generalizability of the pretrained policy network. We run GA in Section 4.2 on these circuits to obtain expert trajectories. The settings of the GA are: population size of 100, gene length of 25, tournament selection (K=5), uniform crossover (35%), random mutation (10%), and 20 iterations. Although pretraining based on these expert trajectories using imitation learning takes time, this is a one-time cost, and LS users are more concerned about the time from design submission to synthesized output, i.e., the exploration time for target designs.

For the BERT model architecture, we adopt the standard BERT-base configuration, which includes 12 transformer layers, a hidden size of 768, and 12 self-attention heads. The GNN component follows the design introduced in Section 4.3.

In the implementation of the AWAC algorithm, we set the following key hyperparameters: the learning rate is set to 3e-4, controlling the update step size for both the policy and value networks; the discount factor γ is set to 0.99 to balance the importance of immediate and future rewards; the soft update coefficient τ is set to 5e-3 to enable gradual updates of the target networks; AWAC λ is set to 1.0, determining the influence of the advantage function during policy updates; and the batch size is set to 2048. To train our RL agents, we use He initialization [39] and Xavier uniform

Table 4. ORL-LO vs. Baselines in 6-LUTs optimization

Benchmark	Initial	resyn2*2	DRiLLS [4]		RL4LS [6]		AlphaSyn [8]		CBTune [7]		Pretrain Only [9]		PIRLLS [9]		ORL-LO (Ours)	
	#LUTs	#LUTs	#LUTs	RT(m)	#LUTs	RT(m)	#LUTs	RT(m)	#LUTs	RT(m)	#LUTs	RT(m)	#LUTs	RT(m)	#LUTs	RT(m)
max	721	719	694	32.58	687.8	54.34	680	5.7	684.25	6.01	688	**0.183**	675.6	2.49	**559.5 ± 0.5**	**8.3 ± 0.2**
adder	249	249	244	24.05	244	10.05	244	6.14	244	5.97	211	**0.184**	196.7	1.62	**191.5 ± 0.4**	**3.2 ± 0.1**
cavlc	116	118	112.2	26.02	111.3	3.22	106	5.35	113	2.37	115	**0.155**	107.6	1.64	**104.6 ± 2.4**	**3.9 ± 0.2**
ctrl	29	29	28	24.25	28	2.85	28	5.68	28	0.59	29	**0.156**	**27**	1.66	**27.7 ± 1.2**	**2.6 ± 0.1**
int2float	47	46	42.6	21.7	42.3	2.81	**39**	5.54	40	2.76	42	**0.165**	**39**	1.54	**38.7 ± 1.4**	**2.8 ± 0.1**
router	73	76	70.1	22.01	69.5	3.07	65	5.34	68.11	2.32	72	**0.16**	61.3	1.64	**59.8 ± 2.2**	**2.8 ± 0.1**
priority	264	220	133.4	23.32	142.9	5.9	135	5.83	138.86	3.41	163	**0.193**	**126.3**	1.58	**130.9 ± 2.5**	**3.2 ± 0.1**
i2c	353	320	292.1	25.17	289.32	7.55	280	6.22	283.11	3.61	299	**0.165**	271.3	1.85	**269.9 ± 1.7**	**2.9 ± 0.1**
sin	1444	1466	1441.5	51.15	1438	20.1	1438	6.77	1441.67	9.71	1450	**0.235**	1434.75	3.57	**1420.0 ± 6.11**	**9.3 ± 0.1**
square	3994	3915	3889.4	130	3889	72.88	3877	8.72	3882.11	25.99	3889	**0.39**	3849.5	7.92	**3726.1 ± 2.1**	**21.4 ± 0.1**
sqrt	8084	5127	4708	147.64	4685.3	196.15	4415	9.83	4607	36.51	4038	**0.47**	3837	17.81	**3009.1 ± 2.8**	**30.6 ± 0.4**
log2	7584	7703	7583.6	198.6	7580.1	125.28	7580	11.78	7580	41.27	7584	**0.61**	7469.6	12.8	**7469.2 ± 5.2**	**31.5 ± 0.1**
multiplier	5678	5713	5678	180.84	5672	187.81	5672	10.34	5679.75	29.08	5675	**0.45**	5663.6	10.57	**5663.3 ± 50.2**	**20.2 ± 0.3**
voter	2744	1828	1834.7	84.43	1678.1	330.48	**1537.4**	7.84	1682.25	11.46	1688	**0.28**	1579	3.7	**1540.3 ± 11.15**	**18.4 ± 0.1**
div	23864	8197	7944.7	259.75	7807.1	482	6650.1	11.88	4180.91	25.58	4124	**0.48**	3963.3	17.52	**3652.4 ± 28.7**	**38.8 ± 0.6**
mem_ctrl	11631	11459	10527.6	229.33	10309.7	1985.84	**9513.2**	10.99	10242.57	45.81	9838	**0.64**	9547.5	17.39	**8151.5 ± 30.7**	**41.7 ± 0.2**
Average	4179.67	2949.06	2826.40	92.55	2792.20	218.14	2641.23	7.75	2555.84	15.78	2494.06	**0.31**	2428.06	6.58	**2250.9 ± 9.8**	**15.1 ± 0.2**

initialization [40] for weight initialization. Following Andrychowicz et al. [41], we scale the weights of the final layer by a factor of 0.01 to prevent bias towards any single action.

We introduce two additional training heuristics to improve sample efficiency: a dynamic episode length mechanism, which terminates an episode early if no area improvement is observed within 5 optimization steps for a given circuit; and an early stopping criterion, which halts the training process if no better area is achieved over 40 consecutive episodes.

The number of 6-LUTs (#LUTs) after technology mapping and the runtime (RT) are evaluated, comparing our framework with baselines such as resyn2, DRiLLS [4], RL4LS [6], AlphaSyn [8], CBTune [7], and our preliminary PIRLLS [9]. For the pretrained policy, actions are selected based on the highest probability for the test circuit state. AWAC is evaluated over ten experiments with different random seeds, averaging the results. To assess its robustness, we report both the mean and standard deviation across multiple independent runs. To match the baselines, EPFL benchmarks [42] are used, and ‘if -a -K 6‘ is applied for 6-LUT optimization.

5.2 Main Results

After pretraining on expert trajectories using seven operators, we refine the action space by selecting operators based on the filtering process described in Section 4.6.2. Additionally, we apply the dynamic synthesis flow length and early stopping mechanism introduced in Section 4.6.3 to further finetune the model. Experiments show that ORL-LO can explore higher-quality optimization flows with lower runtime overhead compared to other methods. As shown in Table 4, ORL-LO outperforms DRiLLS [4], RL4LS [6], AlphaSyn [8], and CBTune [7] by 20.3%, 19.4%, 14.8%, and 13.5% in 6-LUT count after technology mapping, respectively. More importantly, compared to online RL methods such as DRiLLS [4] and AlphaSyn [8], our framework achieves significant speedups of 6.13× and 14.46×. Furthermore, compared to CBTune, it improves optimization quality without significantly increasing runtime. The results show that pretraining the policy network with AWAC and finetuning with multi-processing RL significantly improves optimization efficiency and quality. The GNN-based pretraining phase takes approximately 24 hours to complete, with a GPU memory footprint ranging from 25,000 to 35,000 MB, depending on the size and complexity of the input circuits.

Comparison with Preliminary PIRLLS [9]: We compare our new ORL-LO framework with our preliminary PIRLLS [9], which adopts an approach combining imitation learning for pretraining and an online RL method (PPO) for finetuning. ORL-LO achieves an average 6-LUT reduction of 7.3% compared to PIRLLS, highlighting the effectiveness of our method in optimizing circuit area. The results indicate that our approach, which incorporates offline RL for pretraining, achieves better generalization and finetuning efficiency than methods solely based on imitation learning.

Table 5. Comparison of ORL-LO vs. Baselines in logic level reduction

Benchmark	Initial	resyn2	RL4LS [6]	PIRLLS [9]	ORL-LO
	Level	Level	Level	Level	Level
max	115	**78**	112.02	100	98
adder	121	121	116	**69**	75
cavlc	6	**5**	6.18	6	7
ctrl	**2**	**2**	**2**	**2**	**2**
int2float	**5**	**5**	5.04	**5**	**5**
router	17	8	8.40	**6**	8
priority	101	86	24.88	**28**	**18**
i2c	**7**	8	8.1	8	7
sin	74	76	74.50	72	**73**
square	122	122	121.7	**111**	121
sqrt	3954	2216	1804.66	1883	**1054**
log2	159	153	159	147	**143**
multiplier	**126**	**126**	**126**	**126**	127
voter	37	20	**19.06**	21	20
div	2117	2060	2063.38	2019	**1474**
mem_ctrl	53	51	45.66	45	**37**
Average	438.50	321.06	293.54	290.5	**204.31**

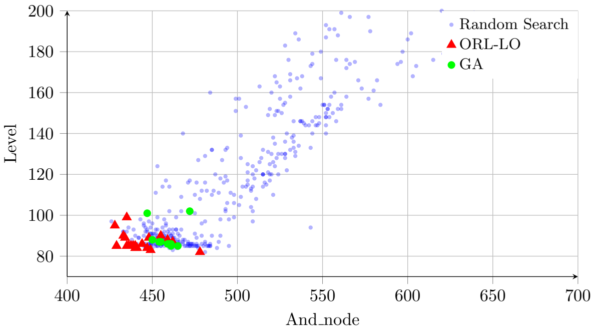


Fig. 5. Multi-objective synthesis with ORL-LO. The optimization results vary by adjusting the weight between `and_node` and `level` in ORL-LO.

The improvement in 6-LUT reduction and runtime efficiency demonstrates the effectiveness of our approach.

Comparison of Logic Level Optimization: Although our objective in this work is circuit area, Table 5 also reports 6-LUT depth for the initial unoptimized mappings, resyn2 mappings, RL4LS [6], PIRLLS [9], and our approach (other methods did not report delay information). We observe that the average 6-LUT level for ORL-LO is 36.4% lower than resyn2, 30.4% lower than RL4LS, and 29.7% lower than PIRLLS [9]. This demonstrates that our approach achieves significant depth reduction compared to all baselines. Importantly, this assures that the area benefits of the proposed RL approach are not coming at the expense of performance.

Multi-objective optimization: Although the baseline methods we selected primarily focus on optimizing a single metric (i.e., circuit area), our proposed ORL-LO framework is not limited to single-objective settings and is inherently extensible to multi-objective optimization. Specifically, the reward function in ORL-LO can be flexibly modified to incorporate multiple objectives such as area, delay, and power. For instance, in Fig. 5, we demonstrate the use of a normalized Area-Delay Product (ADP) as the reward objective. We take the circuit priority as an example to demonstrate the multi-objective optimization capability of ORL-LO. As shown in Fig. 5, 400 randomly generated rewriting sequences are plotted as blue dots, while the red triangles represent the solutions discovered by ORL-LO. In addition, we introduce a GA baseline that applies the same evolutionary framework described in Algorithm 1, using ADP as the fitness function. The GA results are shown as green

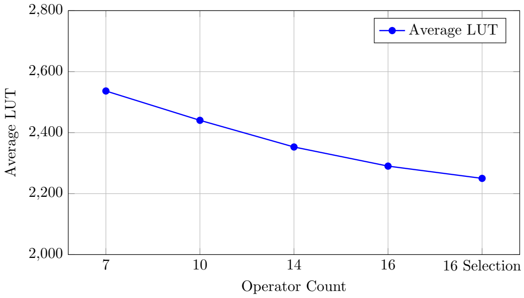


Fig. 6. Impact of operator count.

dots in the figure. Compared to random search and GA, ORL-LO achieves consistently superior trade-offs between area and level, demonstrating its advantage in multi-objective synthesis tasks.

5.3 Impact of Operator Count

Achieving efficient logic optimization requires a careful balance between the number of operators and their selection strategy. Fig. 6 shows the relationship between operator count and 6-LUT reduction. The vertical axis represents the **average 6-LUT count** across the 16 circuits from the EPFL benchmark suite [42], as listed in Table 4, while the horizontal axis denotes the number of operators used for logic restructuring. The label "16 Selection" refers to the method described in Section 4.6.2, where instead of using all 16 operators, an intelligent selection mechanism in Section 4.6.2 identifies the most effective subset. As illustrated in Fig. 6, increasing the number of operators generally leads to better optimization results, as indicated by the decreasing 6-LUT count. This trend suggests that a larger set of operators provides more flexibility in logic restructuring, allowing the model to explore a richer set of transformations that contribute to area optimization.

By applying our operator selection mechanism in Section 4.6.2, we can further refine the operator space for each circuit, tailoring transformations to the specific characteristics of the design. This targeted finetuning improves efficiency while maintaining or even enhancing the quality of results. Notably, the 16-Selection configuration achieves a lower 6-LUT count than the standard 16-operator setup, highlighting the advantage of selecting only the most effective operators. By eliminating redundant transformations, the model focuses on high-impact optimizations, leading to improved convergence and reduced computational overhead. Overall, these results demonstrate that both increasing the number of operators and intelligently selecting them are crucial for achieving high-quality logic optimization.

ORL-LO operates efficiently within a richer and more complex operator space. This expanded search space allows the framework to discover more diverse and effective logic transformations, leading to superior optimization results even under more challenging settings. To ensure a fair comparison with prior methods that adopt only 7 operators, we also evaluate ORL-LO using the same 7-operator set as in the baselines. The results, reported in Table 6, show that ORL-LO continues to outperform all baselines in terms of average 6-LUT count while maintaining competitive runtime. This further highlights the robustness and generalization ability of ORL-LO across both simple and complex operator configurations.

5.4 Ablation Study

We conduct ablation studies to evaluate the impact of different input feature configurations on the learning and optimization performance (Table 7). Each method was trained using 10 parallel sub-environments for 10 episodes, with actor-critic updates from collected trajectories. We compare the following variants: **(i)** "RL": using only scalar circuit features; **(ii)** "RL+His": adding the

Table 6. Average #LUTs and Runtime using 7 operators

Method	Avg. #LUTs	Avg. Runtime (m)
DRiLLS [4]	2826.4	92.6
RL4LS [6]	2792.2	218.1
AlphaSyn [8]	2641.2	7.8
CBTune [7]	2555.8	15.8
PIRLLS [9]	2534.3	6.5
ORL-LO (7 ops)	**2518.3**	**5.92**

Table 7. LUT optimazition results of ablation study

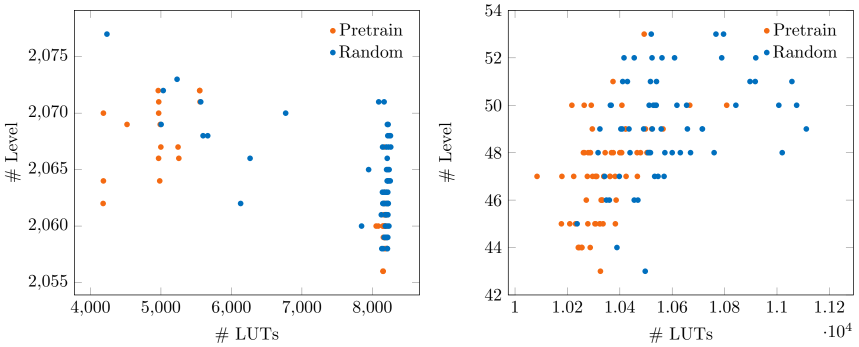
Benchmark	RL	RL+His	RL+BERT		Pretrain+RL+BERT	
	Last	Last	First	Last	First	Last
max	688	691	716	688.1	711	**685.6**
adder	203.1	201	216.6	200.7	211.1	**198.1**
cavlc	111.1	113.9	118.1	111.1	113.9	**110.2**
ctrl	28	28.1	28.8	28	28.2	**27.9**
int2float	43.9	43.4	43.9	**41.4**	43.4	41.7
router	66.5	65.4	70.5	64.4	66.8	**64.2**
priority	142.9	140.7	170.1	135.2	152.5	**127.9**
i2c	289	285.45	299.2	**272.9**	297	274.7
sin	1441.3	1439.7	1452	1440	1445.7	**1438.2**
square	3889	3886.7	3894.6	3862	3889	**3851**
sqrt	4692	4686	4249.7	3870.7	3936	**3847.4**
log2	7489.1	7584	7599.3	7479.1	7548.4	**7471**
multiplier	5687.6	5692	5687.6	5671.4	5692	**5661.6**
voter	1659	1672	1787	1632.4	1716.3	**1611.2**
div	4543	4436	4828	4386.3	4109.7	**4035.9**
mem_ctrl	10291.7	10267.1	10470.5	10274.6	10299.6	**9618.4**
Average	2579	2577	2601.9	2509.8	2516.2	**2441.5**

history of applied operations; **(iii)** "RL+BERT": using pretrained BERT embeddings of flows; **(iv)** "Pretrain+RL+BERT": further incorporating the offline pretrained policy.

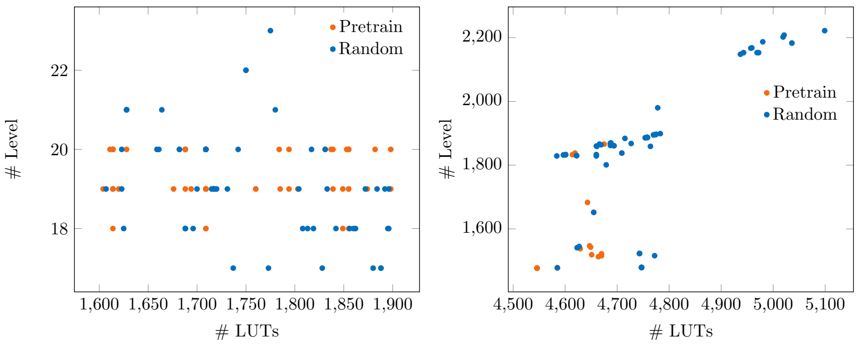
The *First* and *Last* columns in Table 7 report the average best #LUTs achieved in the first and last episodes, respectively. Results show that augmenting scalar features with history and BERT improves optimization quality. The "Pretrain+RL+BERT" model achieves the best overall performance, improving the initial 6-LUT count by 3.3% and the final result by 2.7% compared to "RL+BERT", demonstrating the benefits of offline pretraining. The ablations in Table 7 use the RL setup from our preliminary conference version [9] (imitation learning and on-policy PPO). Although ORL-LO adopts offline RL (AWAC), these results still elucidate the contribution of different state features.

To observe the impact of pretrained policies on RL, we selected four circuits: `div`, `mem_ctrl`, `sqrt`, and `voter`. We compared the first episode optimization results of RL with and without the pretrained policy over 70 runs (Fig. 7). The results show that the pretrained policy consistently provides a better starting point, yielding solutions with lower 6-LUTs and shallower logic depth in the early episodes. Compared to random exploration, the pretrained policy produces denser clusters of high-quality solutions and demonstrates more stable behavior across circuits. This indicates that offline pretraining effectively guides the early-stage search process and improves sample efficiency in circuit optimization tasks.

We also evaluate the contribution of pretrained GNNs by comparing "BERT+GNN" with "BERT-only" during finetuning. Fig. 8 shows the comparison on the `voter` circuit: (a) plots actor loss over training steps, and (b) plots the best achievable area every 80 steps. The GNN-augmented model consistently improves area optimization, escaping local optima that stagnate the BERT-only

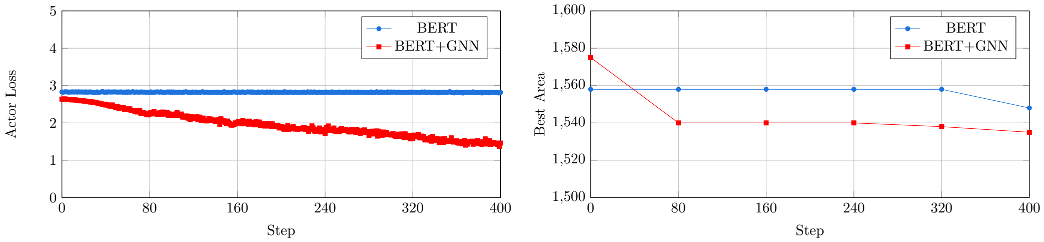


(a) div (b) mem_ctrl



(c) voter (d) sqrt

Fig. 7. Comparison between exploration based on pretraining and random exploration from scratch.



(a) Comparison of actor loss (b) Comparison of best area

Fig. 8. Comparison of BERT+GNN and BERT in terms of best area and actor loss.

baseline. Moreover, the actor loss in BERT+GNN converges faster and stabilizes at a lower value, indicating enhanced representation capacity and more efficient policy learning.

6 CONCLUSIONS

In this paper, we demonstrate that optimal optimization sequences across different circuits exhibit certain similarities. This key observation motivates us to explore methods that can leverage knowledge from previous optimization processes, as traditional models tend to explore these sequences online, ignoring valuable offline information. To address this limitation, we propose ORL-LO, a framework designed to enhance logic optimization in circuit design by utilizing offline RL and finetuning techniques. By pretraining on expert trajectories with the AWAC method, ORL-LO can effectively capture offline knowledge across various designs to learn more generalized policies and handle scenarios not present in expert data. Our framework employs a two-stage approach, combining offline pretraining with online finetuning through AWAC and policy distillation. This approach demonstrates improvements in both optimization quality and exploration efficiency. Experimental results on benchmark circuits suggest that ORL-LO performs competitively with existing methods, achieving reductions in LUT count and runtime. This work illustrates the potential benefits of integrating offline RL into logic synthesis workflows, contributing to more efficient circuit optimization strategies.

ACKNOWLEDGMENT

This work is supported by the National Key R&D Program of China (2022YFB2901100), the National Natural Science Foundation of China (No. 62404021, U22B2038, 62192784), Elite Scientists Sponsorship Program (No. 2023QNRC001) by CAST, the Beijing Natural Science Foundation (No. 4244107, QY24216, QY24204, QY25329), and the State Key Lab of Processors, Institute of Computing Technology, CAS (No. CLQ202504).

REFERENCES

[1] R. Brayton and A. Mishchenko, "ABC: An academic industrial-strength verification tool," in *Computer Aided Verification: 22nd International Conference*, 2010, pp. 24–40.

[2] C. Yu, H. Xiao, and G. De Micheli, "Developing Synthesis Flows Without Human Knowledge," in *ACM/IEEE Design Automation Conference (DAC)*, 2018, pp. 1–6.

[3] A. Basak Chowdhury, B. Tan, R. Carey, T. Jain, R. Karri, and S. Garg, "Bulls-Eye: Active Few-Shot Learning Guided Logic Synthesis," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, vol. 42, no. 8, pp. 2580–2590, 2023.

[4] A. Hosny, S. Hashemi, M. Shalan, and S. Reda, "DRiLLS: Deep reinforcement learning for logic synthesis," in *IEEE/ACM Asia and South Pacific Design Automation Conference (ASPDAC)*, 2020, pp. 581–586.

[5] K. Zhu, M. Liu, H. Chen, Z. Zhao, and D. Z. Pan, "Exploring logic optimizations with reinforcement learning and graph convolutional network," in *ACM/IEEE Workshop on Machine Learning CAD (MLCAD)*, 2020, pp. 145–150.

[6] G. Zhou and J. H. Anderson, "Area-driven FPGA logic synthesis using reinforcement learning," in *IEEE/ACM Asia and South Pacific Design Automation Conference (ASPDAC)*, 2023, pp. 159–165.

[7] F. Liu, Z. Pei, Z. Yu, H. Zheng, Z. He, T. Chen, and B. Yu, "CBTune: Contextual Bandit Tuning for Logic Synthesis," in *IEEE/ACM Proceedings Design, Automation and Test in Europe (DATE)*, 2024, pp. 1–6.

[8] Z. Pei, F. Liu, Z. He, G. Chen, H. Zheng, K. Zhu, and B. Yu, "AlphaSyn: Logic synthesis optimization with efficient monte carlo tree search," in *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2023, pp. 1–9.

[9] G. Dong, J. Zhai, H. Cheng, X. Yang, C. Shi, and K. Zhao, "PIRLLS: Pretraining with Imitation and RL Finetuning for Logic Synthesis," in *IEEE/ACM Asia and South Pacific Design Automation Conference (ASPDAC)*, 2025, pp. 65—-71.

[10] S. Fujimoto, D. Meger, and D. Precup, "Off-Policy Deep Reinforcement Learning without Exploration," in *International Conference on Machine Learning (ICML)*, 2018, pp. 2052–2062.

[11] A. Kumar, J. Fu, M. Soh, G. Tucker, and S. Levine, "Stabilizing Off-Policy Q-Learning via Bootstrapping Error Reduction," in *Annual Conference on Neural Information Processing Systems (NeurIPS)*, 2019, pp. 11 784–11 794.

[12] Y. Wu, G. Tucker, and O. Nachum, "Behavior Regularized Offline Reinforcement Learning," *ArXiv*, 2019.

[13] A. Mishchenko, S. Chatterjee, and R. Brayton, "DAG-aware AIG rewriting: a fresh look at combinational logic synthesis," in *ACM/IEEE Design Automation Conference (DAC)*, 2006, pp. 532–535.

[14] A. Mishchenko and R. K. Brayton, "Scalable Logic Synthesis using a Simple Circuit Structure," in *IEEE/ACM International Workshop on Logic Synthesis*, 2006.

[15] APXML, "Offline RL Distributional Shift," https://apxml.com/zh/courses/advanced-reinforcement-learning/chapter-7-offline-reinforcement-learning/offline-rl-distributional-shift, accessed: 2025-10-24.

[16] R. Huang, A. Geng, and Y. Li, "On the importance of gradients for detecting distributional shifts in the wild," in *Annual Conference on Neural Information Processing Systems (NeurIPS)*, 2021, pp. 677–689.

[17] P. W. Koh, S. Sagawa, H. Marklund, S. M. Xie, M. Zhang, A. Balsubramani, W. Hu, M. Yasunaga, J. Phillips, T. Gao *et al.*, "WILDS: A Benchmark of In-the-Wild Distribution Shifts," in *International Conference on Machine Learning (ICML)*, 2021, pp. 5637–5664.

[18] A. Nair, A. Gupta, M. Dalal, and S. Levine, “AWAC: Accelerating Online Reinforcement Learning with Offline Datasets,” *ArXiv*, 2021.

[19] A. A. Rusu, S. G. Colmenarejo, Çaglar Gülçehre, G. Desjardins, J. Kirkpatrick, R. Pascanu, V. Mnih, K. Kavukcuoglu, and R. Hadsell, “Policy Distillation,” *CoRR*, vol. abs/1511.06295, 2015.

[20] S. Kullback and R. A. Leibler, “On information and sufficiency,” *The Annals of Mathematical Statistics*, vol. 22, no. 1, pp. 79–86, 1951.

[21] D. Sánchez Lopera, L. Servadei, G. N. Kiprit, S. Hazra, R. Wille, and W. Ecker, “A Survey of Graph Neural Networks for Electronic Design Automation,” in *ACM/IEEE Workshop on Machine Learning CAD (MLCAD)*, 2021, pp. 1–6.

[22] S. Yang, Z. Yang, D. Li, Y. Zhang, Z. Zhang, G. Song, and J. Hao, “Versatile multi-stage graph neural network for circuit representation,” in *Annual Conference on Neural Information Processing Systems (NeurIPS)*, 2022, pp. 20 313–20 324.

[23] M. Li, S. Khan, Z. Shi, N. Wang, H. Yu, and Q. Xu, “DeepGate: learning neural representations of logic gates,” in *ACM/IEEE Design Automation Conference (DAC)*, 2022, pp. 667––672.

[24] Z. Shi, H. Pan, S. Khan, M. Li, Y. Liu, J. Huang, H.-L. Zhen, M. Yuan, Z. Chu, and Q. Xu, “DeepGate2: Functionality-Aware Circuit Representation Learning,” in *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2023, pp. 1–9.

[25] J. Liu, J. Zhai, M. Zhao, Z. Lin, B. Yu, and C. Shi, “PolarGate: Breaking the Functionality Representation Bottleneck of And-Inverter Graph Neural Network,” in *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2025, pp. 1–9.

[26] J. Liu, Z. Liu, X. He, J. Zhai, Z. Shi, Q. Xu, B. Yu, and C. Shi, “WideGate: Beyond Directed Acyclic Graph Learning in Subcircuit Boundary Prediction,” in *IEEE/ACM Proceedings Design, Automation and Test in Europe (DATE)*, 2025, pp. 1–7.

[27] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, “BERT: Pre-training of deep bidirectional transformers for language understanding,” in *Annual Conference of the North American Chapter of the Association for Computational Linguistics (NAACL)*, 2018, pp. 4171–4186.

[28] N. Wu, J. Lee, Y. Xie, and C. Hao, “Lostin: Logic optimization via spatio-temporal information with hybrid graph models,” in *2022 IEEE 33rd International Conference on Application-specific Systems, Architectures and Processors (ASAP).* IEEE, 2022, pp. 11–18.

[29] M. Zhao, J. Liu, J. Zhai, and C. Shi, “MILS: Modality Interaction Driven Learning for Logic Synthesis,” in *ACM Great Lakes Symposium on VLSI (GLSVLSI)*, 2025, pp. 64–70.

[30] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba, “OpenAI Gym,” *arXiv: Learning*, Jun 2016.

[31] F. Brglez and H. Fujiwara, “A Neutral Netlist of 10 Combinational Benchmark Circuits and a Target Translator in Fortran,” in *IEEE International Symposium on Circuits and Systems (ISCAS)*, 1985, pp. 677–692.

[32] F. Brglez, D. Bryan, and K. Kozminski, “Combinational profiles of sequential benchmark circuits,” in *IEEE International Symposium on Circuits and Systems (ISCAS)*, May 1989, pp. 1929–1934 vol.3.

[33] F. Corno, M. Reorda, and G. Squillero, “RT-level ITC’99 benchmarks and first ATPG results,” *Design Test of Computers, IEEE*, vol. 17, no. 3, pp. 44–53, Jul 2000.

[34] S. Yang, “Logic Synthesis and Optimization Benchmarks,” 1989 MCNC International Workshop on Logic Synthesis, Tech. Rep., Dec. 1988.

[35] ——, “Logic Synthesis and Optimization Benchmarks User Guide: Version 3.0,” Microelectronics Center of North Carolina (MCNC), Tech. Rep., Jan. 1991.

[36] K. McElvain, “IWLS’93 Benchmark Set: Version 4.0,” a part of IWLS’93 benchmark set, Tech. Rep., May 1993.

[37] C. Albrecht, “IWLS 2005 Benchmarks,” IWLS, Tech. Rep., Jun. 2005.

[38] J. Cong and K. Minkovich, “Optimality Study of Logic Synthesis for LUT-Based FPGAs,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, pp. 230–239, 2007.

[39] K. He, X. Zhang, S. Ren, and J. Sun, “Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification,” in *IEEE International Conference on Computer Vision (ICCV)*, 2015, pp. 1026—-1034.

[40] X. Glorot and Y. Bengio, “Understanding the difficulty of training deep feedforward neural networks,” in *International Conference on Artificial Intelligence and Statistics (AISTATS)*, 2010.

[41] M. Andrychowicz, A. Raichuk, P. Stańczyk, M. Orsini, S. Girgin, R. Marinier, L. Hussenot, M. Geist, O. Pietquin, M. Michalski *et al.*, “What matters in on-policy reinforcement learning? a large-scale empirical study,” *arXiv*, 2020.

[42] L. Amaru, P.-E. Gaillardon, and G. D. Micheli, “The EPFL Combinational Benchmark Suite,” in *IEEE/ACM International Workshop on Logic Synthesis*, Jan 2015.