Parameter Optimization of VLSI Placement Through Deep Reinforcement Learning

Anthony Agnesina, Kyungwook Chang, *Member, IEEE*, and Sung Kyu Lim, *Senior Member, IEEE*

***Abstract*—Critical to achieving power–performance–area goals, a human engineer typically spends a considerable amount of time tuning the multiple settings of a commercial placer. This article proposes a deep reinforcement learning (RL) framework to optimize the placement parameters of a commercial electronic design automation (EDA) tool. We build an autonomous agent that learns to tune parameters without human intervention and domain knowledge, trained solely by RL from self-search. To generalize to unseen netlists, we use a mixture of handcrafted features from graph topology theory and graph embeddings generated using unsupervised graph neural networks. Our RL algorithms are chosen to overcome the sparsity of data and latency of placement runs. As a result, our trained RL agent achieves up to 11% and 2.5% wire length improvements on unseen netlists compared with a human engineer and a state-of-the-art tool auto-tuner in just one placement iteration (20× and 50× fewer iterations). In addition, the success of the RL agent is measured using a statistical test with theoretical guarantees and an optimized sample size.**

***Index Terms*—Deep learning, physical design, very-large-scale integration (VLSI) placement.**

I. Introduction

In RECENT years, the combination of deep learning techniques with reinforcement learning (RL) principles has resulted in self-learning agents achieving superhuman performance in the game of Go, Shogi, and Chess [1]. Deep RL is also used with immense success in real-world applications, such as robotics, finance, and self-driving cars.

The quality of very-large-scale integration (VLSI) placement is essential for the subsequent steps of physical design, with significant repercussions on design quality and design closure. However, recent studies [2] show that existing placers cannot produce near-optimal solutions. The goal of a placement engine is to assign locations for the cells inside the chip's area. The most common target of modern placers is to minimize the total interconnect length, i.e., the estimated half-perimeter wire length (HPWL) from the placed cells' locations.

The algorithms implemented inside the electronic design automation (EDA) tools have parameter settings that users can modify to achieve the desired power–performance–area (PPA). In the authors' experience, more time is spent tuning and running a commercial placer than creating a first version of the design. Tools and flows have increased in complexity, with the modern place and route tools offering more than 10 000 parameter settings. Expert users are required, particularly for the latest technology nodes, with increased cost and risk. Indeed, as the design space of the parameters is too large and complex to be explored by a human engineer alone, one usually relies on expertise and domain knowledge when tuning. However, the correlations between the different parameters and the resulting PPA may be complex or nonintuitive. Placement engines may exhibit nondeterministic behaviors as they heavily rely on handcrafted rules and metaheuristics. Moreover, the advertised goal of a parameter may not directly translate onto the targeted metric.

A state-of-the-art tool auto-tuner [3] widely used in EDA leverages a multiarmed bandit (MAB) to organize a set of classical optimization techniques and efficiently explore the design space. However, these techniques rely on heuristics that are too general and do not consider the specificities of each netlist. Therefore, each new netlist requires to start over parameter exploration. We overcome this limitation in our RL agent by first encoding the netlist information using a mixture of graph handcrafted features and graph neural network (GNN) embeddings. This helps generalize the tuning process from netlist to netlist, saving lengthy and costly placement iterations.

Our RL framework aims to learn an optimization process that finds placement parameters minimizing wire length after placement. The main contributions of this article are as follows.

1) We reduce the significant time expense of VLSI development by applying deep RL to preset the placement parameters of a commercial EDA tool. This is the first work on RL applied to placement parameters optimization to the best of our knowledge.
2) We use a mixture of features relative to topological graph characteristics and graph embeddings generated by a GNN to train an RL agent capable of generalizing its tuning process to unseen netlists.
3) Our RL algorithm overcomes the sparsity of data, and the latency of the design tool runs using multiple environments collecting experiences in parallel. Moreover, this fulfills a multitask learning objective where graph features of a given netlist effectively represent it as a separate task.
4) We build an autonomous agent that iteratively learns to tune parameter settings to optimize the placement without supervised samples. We achieve better wire lengths on unseen netlists than a state-of-the-art auto-tuner, without additional training, and in just one placement iteration.
5) We introduce an existing statistical methodology called the sequential probability ratio test (SPRT), which we use to test a set of specifically tailored hypotheses to measure and benchmark the superiority of our trained RL agent.

Manuscript received 12 July 2021; revised 17 January 2022, 26 April 2022, and 27 June 2022; accepted 7 July 2022. Date of publication 25 July 2022; date of current version 21 March 2023. This work was supported in part by the National Science Foundation under Grant CNS 16-24731, and in part by the Industry Members of the Center for Advanced Electronics in Machine Learning. This article was recommended by Associate Editor I. H.-R. Jiang. *(Corresponding author: Anthony Agnesina.)*

Anthony Agnesina and Sung Kyu Lim are with the Department of Electrical and Computer Engineering, Georgia Institute of Technology, Atlanta, GA 30332 USA (e-mail: agnesina@gatech.edu; limsk@ece.gatech.edu).

Kyungwook Chang is with the Department of Electrical and Computer Engineering, Sungkyunkwan University, Suwon 16419, South Korea (e-mail: k.chang@skku.edu).

Digital Object Identifier 10.1109/TCAD.2022.3193647

II. Related Work

We review the relevant previous works, presenting two approaches to optimize the parameters of an EDA engine, categorized into works on 1) classical hyperparameter/black-box optimization that has extensive study in the literature and 2) a method that views optimization through the lens of learning, which is recent and largely unused in EDA applications.

A. Classical Tuning Methods

Optimizing the parameters of a singular step or the full EDA flow can be seen as an instance of hyperparameter tuning, which has been researched exhaustively in machine learning (ML) due to its extreme importance in the success of neural networks. A metric function f is optimized through parameter search guided by an internal model. Bandit methods minimize the cumulative regret, i.e., the deviation from the optimal value $\mathbb{E}[\sum_{t=1}^{T}(f(\mathbf{x}_\star)-f(\mathbf{x}_t))]$, through a sequence of online function queries. For instance, Ustun *et al.* [4] proposed a multistage FPGA autotuning system by augmenting a bandit framework [3] with a timing regression model to select the future configurations to evaluate. In Bayesian optimization [5], f is assumed to be drawn from a distribution, often a Gaussian process. The posterior built on noisy evaluations of f helps pick the new point to select through optimizing an acquisition function. For example, Ziegler *et al.* [6] used Bayesian learning to build a scalable industrial flow tuning for logic synthesis and physical design, while Kapre *et al.* [7] utilized this strategy to generate good quality FPGA CAD tool parameter sets. Some works, including [6], use precollected offline data to complement these generic optimization methods and improve sampling efficiency. In [8], an XGBoost model is built to learn parameter importance from already synthesized designs. Kwon *et al.* [9] built an online recommender system based on a QoR prediction model trained on archived design data obtained via iterative tuning.

B. Learning to Optimize

On the other side, learning to optimize (L2O) is a metamorphosis of traditional optimization methods, which, with ML, builds new optimizers whose behaviors do not follow a fixed algorithm but are trained on sample data. Therefore, they can achieve faster convergence speeds than classical methods. In this paradigm, the searching point is updated iteratively by $\mathbf{x}_{t+1} = \mathbf{x}_t - g(\mathbf{z}_t, \boldsymbol{\theta})$, where $g(\cdot, \boldsymbol{\theta})$ is the update rule represented by an ML model and $\mathbf{z}_t$ is the state of the optimization process at time t, an ensemble built from previous iterates $\mathbf{z}_t \propto f(\mathbf{x}_{i\le t})$. The weighted sum of the objective function over a timespan is optimized $\sum_{t=1}^{T} w_t f(\mathbf{x}_t)$ to improve the convergence speed. In most cases, $w_t = \mathbb{1}_{\{t=T\}}(t)$.

State-of-the-art methods, mathematical foundations, and critical challenges for L2O research are well summarized in [10]. The best models rely on a recurrent network trained with policy gradient, whose recurrent nature helps mimic the update rule of a sequential gradient-based optimizer. This emerging approach, supplemented with a multitask learning methodology, forms the basis of our work.

III. RL Environment

A. Placement Problem

The purpose of this work is to *indirectly* solve the classical VLSI placement problem, which we formulate as follows.

Classical Placement Problem	
Input	Gate-level netlist, technology, and cell libraries.
Output	Legal placement solution.
Objective	Minimize wire length.
How?	Tune the options of an existing commercial placement engine rather than solve the placement itself.

Placement is one of the most critical steps in VLSI design because it determines the landscape of a silicon chip and heavily influences the subsequent circuit optimizations [11]. However, solving a placement problem instance efficiently as a pure combinational task is difficult *per se*, owing to its NP-completeness. Furthermore, the close interrelation between the many sequential steps of the physical design flow makes obtaining a good placement quality increasingly difficult [12]. For example, placing engines must simultaneously optimize conflicting cost metrics of silicon area, circuit performance, and power without violating on-chip overlap rules. Modern placing engines approach this conundrum using an analytical formulation to co-optimize the wire length subject to density constraints in a single quadratic or nonlinear cost function.

Formally, given a hypergraph representation of the netlist, $G = (V, E)$, where the vertices $V = \{v_1, \ldots, v_n\}$ represent the cells and the hyperedges $E = \{e_1, \ldots, e_m\}$ represent the nets, an analytical placer targets the minimization problem

$$\min_{x,y}\left\{\sum_{e\in E}\mathrm{HPWL}(e; x, y) + \lambda D(x, y)\right\} \tag{1}$$

where

$$\mathrm{HPWL}(e; x, y) = \max_{v_i,v_j\in e}|x_i - x_j| + \max_{v_i,v_j\in e}|y_i - y_j| \tag{2}$$

is the HPWL, (x, y) are the 2-D cell locations, and the density function D ensures no overlaps among cells through the progressive increase of the Lagrange multiplier λ.

In this work, we do not propose a new placer. Instead, appreciating the high-quality placements offered by commercial engines, we propose leveraging them while taking advantage of their sizeable parametrizable space and its opportunity for PPA gains. Traditionally, experienced designers expend significant time searching manually in a wide range of candidate parameters through iterative trial and error to find tool parameter settings that satisfy their PPA goals. However, owing to the vast design space and cost of PPA evaluations from hours to days, only a tiny portion of the possible parameter sets can be explored, resulting in suboptimal designs. Furthermore, due to the latency, complexity, and unpredictability of tool outcomes, traditional black-box optimization methods, such as Simulated Annealing and Genetic Algorithms (GAs), are insufficient to obtain high-quality design implementations. Thus, we propose using RL to automatize the time-consuming tuning of placement parameters and exploit deep ML techniques to generalize the tuning process to unseen netlists.

B. Overview

We build an RL agent that tunes the parameter settings of the placement tool autonomously, intending to minimize wire length. Our RL problem consists of the following four key elements.

1) *States:* The set of all netlists in the world ($\mathcal{N}$) and all possible parameter settings combinations ($\mathcal{P}$) from the placement tool (e.g., Cadence Innovus or Synopsys ICC2). A single state s consists of a unique netlist and a current parameter set.
2) *Actions:* The set of actions that the agent can use to modify the current parameters. An action a changes the setting of a subset of parameters.

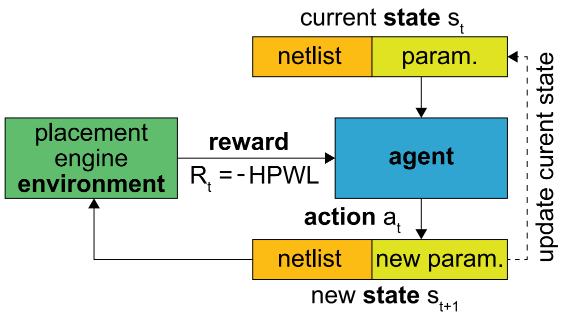

Fig. 1. RL agent–environment interaction in the proposed methodology.

3) *State Transition:* Given a state (s_t) and an action, the next state (s_{t+1}) is the same netlist with updated parameters.
4) *Reward:* Minus the HPWL output from the commercial EDA placement tool. The reward increases if the action improves the parameter settings in terms of minimizing wire length.

As depicted in Fig. 1, in RL, an *agent* learns from interacting with its *environment* over a number of discrete time steps. At each time step t, the agent receives a state s_t and selects an action a_t from a set of possible actions $\mathcal{A}$ according to its policy π, where π maps states to actions. In return, the agent receives a reward signal R_t and transitions to the next state s_{t+1}. This process continues until the agent reaches a terminal state, after which the process restarts. The goal of the agent is to maximize its long-term return

$$G_t = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \tag{3}$$

where γ is a factor discounting future rewards.

An *optimal* policy is one that maximizes the expected returns or *values*. The value function $v_\pi(s_t)$ is the expected return starting from state s_t when following policy π, of the form:

$$v_\pi(s_t) = \mathbb{E}_\pi[G_t | s_t] = \mathbb{E}_\pi\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} | s_t\right]. \tag{4}$$

C. Our RL Settings

We formulate the placement parameters optimization task led by an RL agent as follows.

RL Placement Parameter Optimization Problem	
Goal	Given a netlist, find $\arg\min_{p \in \mathcal{P}}$ HPWL(p) where $\mathcal{P}$ is the set of all parameter combinations and HPWL is given by the tool.
How?	1) Define Environment as placement tool black-box. 2) Define state $s \approx$ current set of parameters $p_{curr} \in \mathcal{P}$ and target netlist. 3) Define actions to modify p_{curr}. 4) Define a reward $R \propto -$HPWL so that the agent's goal is to decrease wire length. 5) Select a discount factor $\gamma < 1$ to force the agent to reduce wire length in as few steps as possible.

This is a combinatorial optimization problem where $\mathcal{P}$ is very large and exhaustive search is infeasible.

We now present a more formal definition of the problem at hand and our proposed approach. Given a netlist n, we wish to find $p^* = \arg\max_{p \in \mathcal{P}} \text{PPA}_{\text{tool}}(n, p)$, where $\mathcal{P}$ is the space of all parameter sets, and PPA_{tool} is the EDA tool output. In this work, the latter corresponds to $-$HPWL but could be any metric of interest in all generality. The black-box function PPA_{tool} is not directly available to the learner and can only be evaluated at a query point (n, p). These queries are expensive in terms of runtime and computing resources.

To find p^*, we first define $s_t = (n, p_t)$ as the state at timestep t ($\equiv$ query). An action a_t turns p_t into p_{t+1}. The environment is then queried by running the commercial tool with parameters p_{t+1} for netlist n. The reward is returned as R_t, a stochastic evaluation of $\text{PPA}_{\text{tool}}(n, p_{t+1})$. The new state is now $s_{t+1} = (n, p_{t+1})$. After L evaluations, we output our best guess $p^{\text{Guess}} = p(L)$, which could be different from p_L.

We posit that this modified problem can be reduced to learning a sequential optimizer represented as an optimal policy π^* prescribing the optimal action to take in a specific state. The optimizer should learn the optimization process for all possible netlists $\mathcal{N}$. Given a new netlist n', an optimal parameter set is obtained as

$$p^*(n') = a_L\big(a_{L-1}(\cdots a_0(p_0)\ \ldots)\big) \tag{5}$$

where $a_i = \pi^*(n', p_i)$. In that context, a policy π is considered optimal if it maximizes the expectation

$$\mathbb{E}_{n \sim \mathcal{N}}\Big[\text{PPA}_{\text{tool}}(n, p^{\text{Guess}})\Big] \tag{6}$$

where p^{Guess} is found on the trajectory following π.

For an agent to correctly select an action, we must first define a good representation of its environment. In our case, the representation of the environment is given by a human expert as presented.

D. Our States

We define our state as the joint values of 12 placement parameters from Cadence Innovus used to perform the current placement (Table I) and information metrics on the netlist being placed. The netlist information consists of a mixture of metadata knowledge (e.g., number of cells and floorplan area) with topological graph features (Table II) and unsupervised features extracted using a GNN. Netlist characteristics are essential to transfer the knowledge across very different netlists so that our agent generalizes its tuning process to unseen netlists. Indeed, the optimal policy is likely related to each netlist's particularities. Thus, our state can be written as a concatenation of one-hot encoded categorical parameters (Booleans or enumerates), integer parameters, and integer and float netlist features.

We carefully selected the placement parameters among the 60 available ones in the software. We pruned the ones not relevant to our study; we do not, for example, consider structured data paths, fillers, scan chains, shifters, and IR drops. Our goal is a proof of concept and to show RL methods' applicability to the design space exploration of commercial VLSI tools knobs. Thus, we picked the 12 most common parameters understood even by inexperienced designers. Moreover, the small number of these makes action definition simpler, while a large number of these would result in a much larger space and an impractical definition of actions for our proof-of-concept purpose.

We transform the synthesized netlists into graphs using the model shown in Fig. 2. The resulting abstract graph $G = (V, E)$ is as follows. For each net e in the netlist with driving cell s_e and load cells $\{t_0, \ldots, t_n\}_e$, we create a directed edge in G per pair (s_e, t_i). This edge-based representation versus hyperedge-based makes applying standard graph algorithms to the netlist easy. The directed nature of the graph represents the natural flow of electrical signals of the nets

TABLE I
12 TARGETED PLACEMENT PARAMETERS. THE SOLUTION SPACE IS $101^3 \cdot 3^6 \cdot 2^3 = 6 \times 10^9$

Name	Objective	Type	Groups	# val
eco max distance	maximum distance allowed during placement legalization	integer	detail	[0, 100]
legalization gap	minimum sites gap between instances	integer	detail	[0, 100]
max density	controls the maximum density of local bins	integer	global	[0, 100]
eco priority	instance priority for refine place	enum	detail	3
activity power driven	level of effort for activity power driven placer	enum	detail + effort	3
wire length opt	optimizes wire length by swapping cells	enum	detail + effort	3
blockage channel	creates placement blockages in narrow channels between macros	enum	global	3
timing effort	level of effort for timing driven placer	enum	global + effort	2
clock power driven	level of effort for clock power driven placer	enum	global + effort	3
congestion effort	the effort level for relieving congestion	enum	global + effort	3
clock gate aware	specifies that placement is aware of clock gate cells in the design	bool	global	2
uniform density	enables even cell distribution	bool	global	2

TABLE II
OUR 20 HANDCRAFTED NETLIST FEATURES

Metadata (10)		Topological (10)	
Name	Type	Name	Type
# cells	integer	average degree	float
# nets	integer	average fanout	float
# cell pins	integer	largest SCC	integer
# IO	integer	max. clique	integer
# nets w. fanout $\in]5, 10[$	integer	chromatic nb.	integer
# nets w. fanout ≥ 10	integer	max. logic level	integer
# FFs	integer	RCC	float
total cell area (μm^2)	integer	$\overline{CC}$	float
# hardmacros	integer	Fiedler value	float
macro area (μm^2)	integer	spectral radius	float

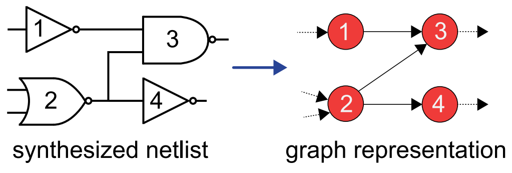

Fig. 2. Our netlist to graph transformation.

from source to sinks. The directed nature of the graph is necessary for the correct definition of graph characteristics proposed, such as the logic levels. However, this representation is switched to undirected when needed by other graph algorithms.

We propose two ways to extract meaningful features from this graph representation.

1) Topological Graph Features: We borrow concepts from graph theory to learn rich graph representations that uniquely classify each graph. We use Boost and BLAS optimized C++ libraries to extract our features efficiently on large graphs—collected in less than 1 h for the larger netlists. Let $G = (V, E)$ be the directed cyclic graph representing the netlist obtained using the above model. Global signals, such as reset, clock, or VDD/VSS, are not considered when building the graph. Multiple connections between two vertices in the graph are merged, and self-loops are eliminated. We consider the following graph features to capture the netlist's complex topology characteristics (e.g., connections and spectral).

1) *Strongly Connected Components (SCCs):* A strong component of G is a subgraph S of G if for any pair of vertices $u, v \in S$ there is a directed cycle in S containing u and v. We compute them in linear time using directed a depth-first search with an algorithm due to Tarjan [13].
2) *Clique:* Given an integer m, does G contains K_m (the complete graph on m vertices) as a subgraph? We use the Bron–Kerbosch algorithm [14] to find the maximal cliques.
3) *k-Colorability:* Given an integer k, is there a coloring of G with k or fewer colors? A *coloring* is a map $\chi : V \to C$ such as two adjacent vertices have the same color; i.e., if $(u, v) \in E$, then $\chi(u) \neq \chi(v)$. Minimum k (the chromatic number) is computed using a technique proposed in [15].
4) *Logic Levels:* What is the maximum distance (# gates traversed) between two flip-flops? $LL = \max_{a,b \in FFs} d(a, b)$.
5) *Rich Club Coefficient:* How well do high-degree (rich) nodes know each other? Let $G_k = (V_k, E_k)$ be the filtered graph of G with only nodes of degree $> k$, then $\text{RCC}_k = ([2|E_k|]/[|V_k|(|V_k| - 1)])$ [16].
6) *Clustering Coefficient:* A measure of the cliquishness of nodes neighborhoods [17] of the form

$$\overline{CC} = \frac{1}{|V|} \sum_{i \in V} \frac{|e_{jk} : v_j, v_k \in \text{Neighbors}(i), e_{jk} \in E|}{\deg(i)(\deg(i) - 1)}. \quad (7)$$

7) *Spectral Characteristics:* Using the implicitly restarted Arnoldi method [18], we extract from the Laplacian matrix of G the Fiedler value (second smallest eigenvalue) deeply related to the connectivity properties of G, as well as the spectral radius (largest eigenvalue) relative to the regularity of G.

These features give crucial information about the netlist. For example, connectivity features, such as SCC, maximal clique, and RCC, are essential to capture congestion considerations (considered during placement refinement), while logic levels indirectly translate the difficulty of meeting timing by extracting the longest logic path.

2) Features From Graph Neural Network: Starting from simple node features, including degree, fanout, area, and encoded gate type, we generate node embeddings ($\text{ENC}(v)$) using unsupervised *GraphSAGE* [19] with convolutional aggregation, dropout, and output size of 32. The GNN algorithm iteratively propagates information from a node to its neighbors. The GNN is trained on each graph individually. Then, the graph embedding ($\text{ENC}(G)$) is obtained from the node embeddings with a permutation invariant aggregator, as shown in Fig. 3

$$\text{ENC}(G) = \text{MEAN}(\text{ENC}(v) | v \in V) \quad (8)$$

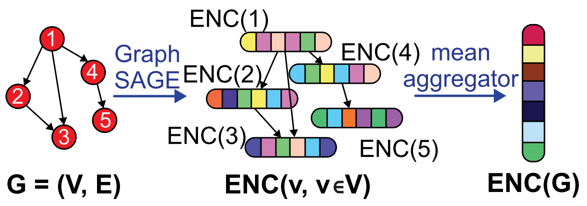

Fig. 3. Graph embedding using a GNN package GraphSAGE [19]. We first extract 32 features for each node in the graph. Next, we calculate the mean among all nodes for each feature, resulting in 32 features for the entire graph.

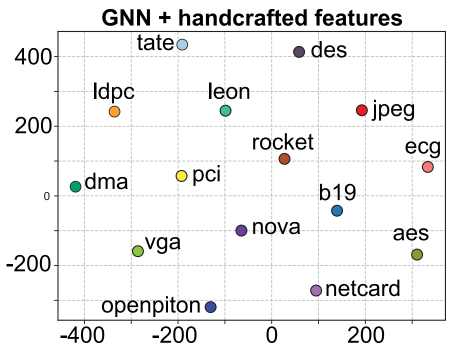

Fig. 4. t-SNE visualization [20] of our 20 handcrafted plus 32 GNN features combined. Points representative of each netlist are well separated, proving graph features capture the differences between netlists well.

where the individual node embeddings are obtained after K recurrent updates

$$\begin{cases} x_v^{(0)} = (\text{degree}(v), \text{fanout}(v), \text{area}(v), \text{type}(v), \dots,) \\ x_v^{(k+1)} = \Psi(x_v^{(k)}, \rho(\{x_u^{(k)} : e = (v, u) \in E\})) \\ \text{ENC}(v) = x_v^{(K)}. \end{cases} \quad (9)$$

Function ρ aggregates features of neighboring nodes, and function Ψ concatenates the node's current representation with the aggregated neighborhood information, which it then nonlinearizes and normalizes.

The t-SNE projection in two dimensions [20] of the vector of graph features is displayed in Fig. 4. We see that all netlists' points are far apart, indicating that the combination of our handcrafted and learned graph features distinguishes well the particularities of each netlist.

E. Our Actions

The state-of-the-art parameter auto-tuner presented in [3] uses a MAB whose arms are classical search techniques, such as Simulated Annealing and GAs. Each technique modifies parameter sets differently, while the MAB algorithm tries to determine the arm with the highest expected reward quickly. Parameter search is done by selecting a search technique one at a time, where the selection is based on a credit assignment score that mixes exploration and exploitation.

We considered the possibility of using these search arms as actions in our RL framework. The problem with this option is that a fundamental assumption in RL tasks is that the Markov property of memorylessness holds; i.e., the optimal action should depend on the last state alone. This assumption is not valid for arms such as GA or any technique with population-based improvements which have a current state depending on all prior states. Instead, they memorize a population of one, usually the best, or multiple parameter settings to create a new candidate setting. In principle, this representation can be made Markovian by aggregating all parameters evaluated into one state. However, this aggregation can produce a state representation impractically too large.

We thus choose to define our own deterministic actions to change the setting of a subset of parameters. They render the state Markovian, i.e., given state–action pair (s_t, a_t), the resulting state s_{t+1} is unique. An advantage of fully observed determinism is that it allows *planning*. Starting from state s_0 and following a satisfactory policy π, the trajectory

$$s_0 \xrightarrow{\pi(s_0)} s_1 \xrightarrow{\pi(s_1)} \cdots \xrightarrow{\pi(s_{n-1})} s_n \quad (10)$$

leads to a parameter set s_n of good quality. If π has been learned, s_n can be computed directly in $\mathcal{O}(1)$ time without performing any placement.

Defining only two actions per placement parameter would result in 24 different actions, too many for the agent to learn well. Thus, we first decided to group variables per type (Boolean, Enumerate, Numeric) and per placement "focus" (Global, Detailed, Effort). Then, for each group, we define expressive yet straightforward actions such as FLIP for Booleans. For enumerates, DOWN $\equiv$ "pass from high to medium," for example. For an integral parameter value x defined in a range $[a, b]$, the updated value of actions UP/DOWN is obtained with the simple transformation $x' = \min(\max(x \pm (b-a)\Delta x_0, a), b)$, where Δx_0 is a fixed delta $\in [0, 1]$ defined per parameter. In the experiments, $\Delta\{\text{eco max distance}\} = 0.10$, $\Delta\{\text{legalization gap}\} = 0.10$, and $\Delta\{\text{max density}\} = 0.03$. The smaller delta for max density accounts for its larger effect, observed empirically, on the HPWL.

We also add one arm that does not modify the current set of parameters. Instead, it triggers an environment's reset if it gets picked multiple times in a row. This leads to the 11 different actions $\mathcal{A}$ presented in Table III. Our action space is designed to be as simple as possible to help neural network training but also expressive enough so that such transformations can reach any parameter settings.

TABLE III
OUR 11 ACTIONS

1. FLIP Booleans
2. UP Integers
3. DOWN Integers
4. UP Efforts
5. DOWN Efforts
6. UP Detailed
7. DOWN Detailed
8. UP Global (does not touch the bool)
9. DOWN Global (does not touch the bool)
10. INVERT-MIX timing vs. congestion vs. WL efforts
11. DO NOTHING

F. Our Reward Structure

To learn with a single RL agent across various netlists with different wire lengths, we cannot define a reward directly linear with HPWL. Thus, to help convergence, we adopt a normalized reward function that renders the magnitude of the value approximations similar among netlists, of the form

$$R_t := \frac{\text{HPWL}_{\text{Human Baseline}} - \text{HPWL}_t}{\text{HPWL}_{\text{Human Baseline}}}. \quad (11)$$

While defining rewards in this manner necessitates knowing $\text{HPWL}_{\text{Human Baseline}}$, an expected baseline wire length per design, this only requires one placement to be completed by an engineer.

IV. RL Placement Agent

A. Overview

Using the definition of the environment presented in the previous section, we train an agent to tune the parameters of the placement tool autonomously. Here is our approach.

1) The agent learns the optimal action for a given state. This action is chosen based on its policy network probability outputs.
2) We adopt an *actor–critic* framework to train the policy network effectively, which brings the benefits of value-based and policy-based optimization algorithms together.
3) To solve the well-known shortcomings of RL in EDA of latency and sparsity, we implement multiple environments collecting different experiences in parallel.
4) Our agent architecture utilizes a deep neural network comprising a recurrent layer with an attention mechanism to enable learning a recursive optimization process with complex dependencies.

B. Goal of Learning

We choose to use policy-based RL from the many ways of learning how to behave in an environment. We state the formal definition of this problem as follows.

Policy Based RL Problem	
Goal	Learn the optimal policy $\pi^*(a\|s)$
How?	1) Approximate a policy by parameterized $\pi_{\boldsymbol{\theta}}(a\|s)$. 2) Define objective $J(\boldsymbol{\theta}) = \mathbb{E}_{\pi_{\boldsymbol{\theta}}}[v_{\pi_{\boldsymbol{\theta}}}(s)]$. 3) Find $\arg\max_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$ with Stochastic Gradient.

This optimization problem aims to learn which action a to take in a specific state s. We represent the parametrized policy $\pi_{\boldsymbol{\theta}}$ by a deep neural network. The main reasons for choosing this framework are as follows.

1) It is model-free, which is essential as the placer tool environment is complex and challenging to model.
2) Our intuition is that the optimal policy may be simple to learn and represent (e.g., keep increasing the effort), while the value of a parameter setting may not be trivial or change significantly based on observation.
3) Policy optimization often shows good convergence properties.
4) A tabular version storing every state–action pair (s_t, a_t) and reward R_t would be too large to store in memory and too slow to learn the value of each state individually.
5) A neural network is a universal approximator that has been shown to generalize well when the experience is not independent and identically distributed (i.i.d.) and nonstationary (successive states are correlated).

C. How to Learn: The Actor–Critic Framework

In our chosen architecture, we learn a policy that optimizes the value while learning the value simultaneously. For learning, it is often beneficial to use as much knowledge observed from the environment as possible and hang off other predictions rather than solely predicting the policy. This type of framework, called *actor–critic*, is shown in Fig. 5. The policy is known as the actor because it is used to select actions, and the estimated value function is known as the critic because it criticizes the actions made by the actor.

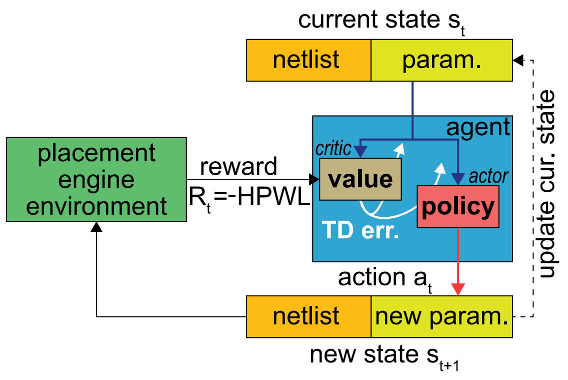

Fig. 5. Actor–critic framework. The critic learns about and critiques the actor's current policy.

Actor–critic algorithms combine *value-based* and *policy-based* methods. Value-based algorithms learn to approximate values $v_{\mathbf{w}}(s) \approx v_\pi(s)$ by exploiting the Bellman equation

$$v_\pi(s) = \mathbb{E}\big[R_{t+1} + \gamma v_\pi(s_{t+1}) | s_t = s\big] \tag{12}$$

which is used in temporal difference (TD)

$$\Delta \mathbf{w}_t = \big(R_{t+1} + \gamma v_{\mathbf{w}}(s_{t+1}) - v_{\mathbf{w}}(s_t)\big) \nabla_{\mathbf{w}} v_{\mathbf{w}}(s_t). \tag{13}$$

On the other hand, policy-based algorithms update a parameterized policy $\pi_{\boldsymbol{\theta}}(a_t|s_t)$ directly through stochastic gradient ascent in the direction of the value with

$$\Delta \boldsymbol{\theta}_t = G_t \nabla_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(a_t|s_t). \tag{14}$$

In *actor–critic*, the policy updates are computed from incomplete episodes by using truncated returns that bootstrap on the value estimate at state s_{t+n} according to $v_{\mathbf{w}}$, given by

$$G_t^{(n)} = \sum_{k=0}^{n-1} \gamma^k R_{t+k+1} + \gamma^n v_{\mathbf{w}}(s_{t+n}). \tag{15}$$

This reduces the variance of the updates and propagates rewards faster. The variance can be further reduced using state-values as a baseline in policy updates, as in *advantage* actor–critic updates of the form

$$\Delta \boldsymbol{\theta}_t = \Big(G_t^{(n)} - v_{\mathbf{w}}(s_t)\Big) \nabla_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(a_t|s_t). \tag{16}$$

The critic updates parameters $\mathbf{w}$ of $v_{\mathbf{w}}$ by n-step TD following (13), and the actor updates parameters $\boldsymbol{\theta}$ of $\pi_{\boldsymbol{\theta}}$ in the direction suggested by the critic by policy gradient with (16). In this work, we use the *advantage actor–critic* method, A2C [21], which produced excellent results in diverse environments. As shown in (16), an *advantage* function formed as the difference between returns and baseline state–action estimate is used instead of raw returns. The *advantage* can be considered a measure of how good a given action is compared to some average.

D. Synchronous Actor/Critic Implementation

The main issues plaguing the use of RL in EDA are the latency of tool runs (it takes minutes to hours to perform one placement) and the sparsity of data (there is no database of millions of netlists placed designs or layouts). We implement a parallel version of A2C to solve both issues, as depicted in Fig. 6. In this implementation, an agent learns from the experiences of multiple *Actors* interacting

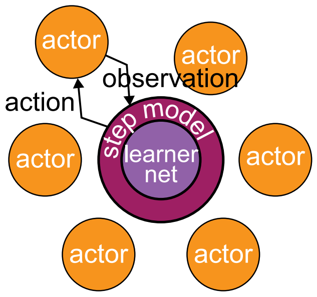

Fig. 6. Synchronous parallel learner. The global network sends actions to the actors through the step model. Each actor gathers experiences from their own environment.

in parallel with their copy of the environment. This configuration increases the throughput of acting and learning and helps decorrelate samples during training for data efficiency [22].

The learning updates may be applied synchronously or asynchronously in parallel training setups. We use a synchronous version, i.e., a deterministic implementation that waits for each Actor to finish its segment of experience (according to the current policy provided by the step model) before performing a single batch update to the network weights. One advantage is that it provides larger batch sizes, which are more effectively used by computing resources.

The parallel training setup does not modify the equations presented before. Instead, the gradients are just accumulated among all the environments' batches.

E. Two-Head Network Architecture

The *actor–critic* framework uses both policy and value models. The full agent network can be represented as a deep neural network $(\pi_{\boldsymbol{\theta}}, v_{\mathbf{W}}) = f(s)$. This neural network takes the state $s = (p \circ n)$ made of parameter values p and netlist features n and outputs a vector of action probabilities with components $\pi_{\boldsymbol{\theta}}(a)$ for each action a, and a scalar value $v_{\mathbf{W}}(s)$ estimating the expected cumulative reward G from state s.

The policy tells us *how to modify a placement parameter setting*, and the value network tells us *how good this current setting is*. We share the body of the network to allow value and policy predictions to inform one another. The parameters are adjusted by gradient ascent on a loss function that sums over the losses of the policy and the value plus a regularization term, whose gradient is defined as in [23]

$$\begin{gathered}\underbrace{\left(G_t^{(n)} - v_{\mathbf{W}}(s_t)\right)\nabla_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(a_t|s_t)}_{\text{policy gradient}} + \\ \beta \underbrace{\left(G_t^{(n)} - v_{\mathbf{W}}(s_t)\right)\nabla_{\mathbf{W}} v_{\mathbf{W}}(s_t)}_{\text{value estimation gradient}} + \\ \underbrace{\eta \sum_a \pi_{\boldsymbol{\theta}}(a|s_t) \log \pi_{\boldsymbol{\theta}}(a|s_t)}_{\text{entropy regularization}}.\end{gathered} \tag{17}$$

The entropy regularization pushes entropy up to encourage exploration, and β and η are hyperparameters that balance the importance of the loss components.

The complete architecture of our deep neural network is shown in Fig. 7. For value and policy predictions, the concatenation of placement parameters with graph extracted features is first passed through two feedforward fully connected (FC) layers with tanh activations, followed by an FC linear layer. This is followed by a long

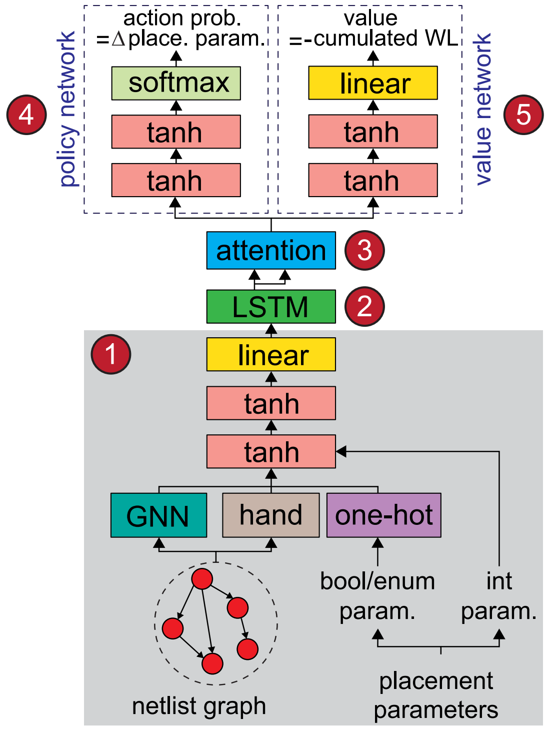

Fig. 7. Overall network architecture of our agent. Combining an LSTM with an attention mechanism enables learning a complex recurrent optimization process. Table IV provides the details of the subnetworks.

short-term memory (LSTM) module with layer normalization and 16 hidden standard units with Forget gate. The feedforward FC layers have no memory. The model can base its actions on previous states by introducing an LSTM in the network, which is a recurrent layer. This is motivated by the fact that traditional optimization methods are based on recurrent approaches. Moreover, we add a sequence-to-one global attention mechanism [24] inspired by state-of-the-art natural language processing architectures to help the recurrent layer (RNN) focus on essential parts of the recursion. Let $\boldsymbol{h}_t$ be the hidden state of the RNN. Then, the attention alignment weights $\boldsymbol{a}_t$ with each source hidden state $\bar{\boldsymbol{h}}_s$ are defined as

$$\boldsymbol{a}_t(s) = \frac{\exp\left(\text{score}(\boldsymbol{h}_t, \bar{\boldsymbol{h}}_s)\right)}{\sum_{s'} \exp\left(\text{score}(\boldsymbol{h}_t, \bar{\boldsymbol{h}}_{s'})\right)} \tag{18}$$

where the alignment score function is

$$\text{score}(\boldsymbol{h}_t, \bar{\boldsymbol{h}}_s) = \boldsymbol{h}_t^\top \boldsymbol{W}_{\boldsymbol{a}} \bar{\boldsymbol{h}}_s. \tag{19}$$

The global context vector

$$\boldsymbol{c}_t = \sum_s \boldsymbol{a}_t(s) \bar{\boldsymbol{h}}_s \tag{20}$$

is combined with the hidden state to produce an attentional hidden state, yielding

$$\tilde{\boldsymbol{h}}_t = \tanh(\boldsymbol{W}_{\boldsymbol{c}}[\boldsymbol{c}_t \circ \boldsymbol{h}_t]). \tag{21}$$

This hidden state is then fed to the two heads of the network, composed of two FC layers with an output softmax layer for the policy and a linear output layer for the value. The parameters of our network are summarized in Table IV.

TABLE IV
NEURAL NETWORK PARAMETERS USED IN OUR RL AGENT ARCHITECTURE IN FIG. 7. THE NUMBER OF INPUTS OF THE FIRST FC LAYER IS AS FOLLOWS: 32 FROM GNN, 20 FROM TABLE II, 24 ONE-HOT ENCODING FOR THE ENUM/BOOL TYPES FROM TABLE I, AND 3 INTEGER TYPES FROM TABLE I

Part	Input	Hidden	Output
1. Shared Body	79	(64, 32) (tanh)	16 (linear)
2. LSTM (6 unroll)	16	16	16×6
3. Attention	16×6	$\boldsymbol{W_a}, \boldsymbol{W_c}$	16
4. Policy	16	(32, 32) (tanh)	11 (softmax)
5. Value	16	(32, 16) (tanh)	1 (linear)

F. Our Self-Play Strategy

Inspired by AlphaZero [1], our model learns without supervised samples. We do not use expert knowledge to pretrain the network using well-known parameter sets or actions. While the agent makes random moves at first, the idea is that by relying on zero human bias, the agent may learn counter-intuitive moves and achieve superhuman tuning capabilities.

G. Multitask Learning

We adopt a multitask learning framework during training to build an agent that can perform well on multiple netlists. We posit that each netlist corresponds to a different *task* to follow this paradigm. In principle, different VLSI netlists share the same overall structure (≈ directed cyclic graph with small-degree nodes), making multitask learning more efficient and learning all the tasks more quickly or proficiently than learning them independently. This principle will also allow generalization to unseen netlists, whose structures are similar to the trained ones. While there are various multitask learning methodologies [25], we use one of the most direct approaches: aggregate the data across tasks and learn a single model. Thus, our vanilla multitask learning objective is

$$\min_{\boldsymbol{\theta}} \sum_{t=1}^{T} \mathcal{L}_t(\boldsymbol{\theta}, D_t) \tag{22}$$

where each loss $\mathcal{L}_t$ is computed with (17), and the task t (= netlist n) is specified as an extra set of features in the state definition $s = (n, p)$. D_t corresponds to the data obtained for task t. This *concatenation-based* conditioning is standard but expressive enough to dilute the netlist information inside the network. The reward function also depends on the task $R = R(n)$, as highlighted by the HPWL baseline defined per netlist in (11). Finally, we solve the reduced single-task learning problem with our parallel A2C: scattering the T netlists over the different actors will help optimize the above global loss.

V. EXPERIMENTAL RESULTS

To train and test our agent, we selected 15 benchmarks designs from OpenCores, ISPD 2012 contest, and two RISC-V single cores, presented in Table V. We use the first 11 designs for training, then the following LDPC design is held-out to define a parameter set selection criterion post-training, and finally, the last three unseen designs are used for actual testing. We synthesize the RTL netlists using Synopsys Design Compiler. We use TSMC 28-nm technology node. The placements are done with Cadence Innovus 17.1. The aspect ratio of the floorplans is fixed to 1, and appropriate fixed clock frequencies are selected. Memory macros of RocketTile and OpenPiton Core benchmarks are preplaced by hand. A lower bound of total cell area divided by floorplan area is set on parameter *max density* for successful placements. IO pins are placed automatically by the tool between metals 4 and 6.

TABLE V
BENCHMARK STATISTICS BASED ON A COMMERCIAL 28-NM TECHNOLOGY. RCC IS THE RICH CLUB COEFFICIENT (e^{-4}), LL IS THE MAXIMUM LOGIC LEVEL, AND SP. R. DENOTES THE SPECTRAL RADIUS. RT IS THE AVERAGE PLACEMENT RUNTIME USING INNOVUS (IN MINUTES)

Name	#cells	#nets	#IO	RCC_3	LL	Sp. R.	RT
training set							
PCI	1.2K	1.4K	361	510	17	25.6	0.5
DMA	10K	11K	959	65	25	26.4	1
B19	33K	34K	47	19	86	36.1	2
DES	47K	48K	370	14	16	25.6	2
VGA	52K	52K	184	15	25	26.5	3
ECG	83K	84K	1.7K	7.5	23	26.8	4
Rocket	92K	95K	377	8.1	42	514.0	6
AES	112K	112K	390	5.8	14	102.0	6
Nova	153K	155K	174	4.6	57	11,298	9
Tate	187K	188K	1.9K	3.2	21	25.9	10
JPEG	239K	267K	67	2.8	30	287.0	12
selection criterion definition set							
LDPC	39K	41K	4.1K	18	19	328.0	2
test set (unseen netlist)							
OpenPiton	188K	196K	1.6K	3.9	76	3940	19
Netcard	300K	301K	1.8K	2.9	32	27.3	24
Leon3	326K	327K	333	2.4	44	29.5	26

A. RL Network Training Setting

We define our environment using the OpenAI Gym interface [26] and implement our RL agent network in Tensorflow. We use 16 parallel environments (16 threads) in our synchronous A2C framework. We use the smallest netlists during training for the following reason. The synchronous nature of A2C demands all individual episodes to be completed for all environments before the weight update is possible, i.e., an entire batch is collected. As netlists are randomly and uniformly assigned when environments are reset, and the training spans numerous resets, $\mathbb{E}[\text{\#it. per netlist}]$ is fairly constant across all netlists. So, the total makespan, here the training runtime, is equal to the makespan of the longest netlist

$$\mathbb{E}[\text{makespan}] = \underbrace{\mathbb{E}[\text{\#it. per netlist}]}_{\text{cst.}} \times \max_{n \in \mathcal{N}} \mathbb{E}[\,\text{RT}(n)]. \tag{23}$$

This equation motivates the use of smaller netlists whose size correlates well with the placement runtime, as shown in the last column of Table V, to speed up training.

We perform tuning of the hyperparameters of our network using Bayesian optimization, which results in stronger agents. The learning curve of our A2C agent in our custom Innovus environment is shown in Fig. 8. We observe that the mean reward across all netlists converges asymptotically to a value of 6.8%, meaning wire length is reduced on average by 6.8%.

Training over 150 updates (×6 iterations = 14 400 placements) takes about 100 h. Note that 99% of that time is spent on performing the placements, while updating the neural network weights takes less

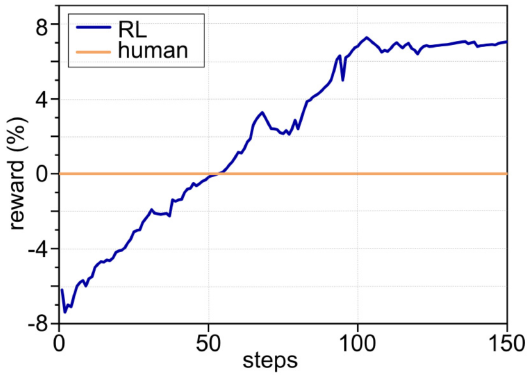

Fig. 8. Training our agent for 150 updates ($\times$ 16 $\times$ 6 = 14 400 placements). The reward is an aggregate of all training netlists' rewards. Training time is within 100 h. Human baseline: reward = 0.

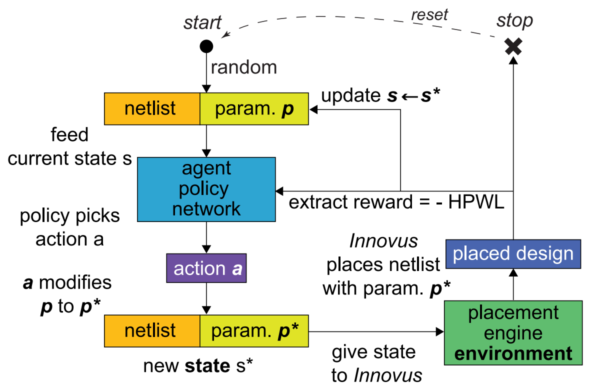

Fig. 9. Our RL training workflow in each environment.

than 20 min. Without parallelization, training over the same number of placements would take 16 $\times$ 100 h = 27 days.

An explained variance of 0.67 shows that the value function explains relatively well the observed returns. We use a discount factor $\gamma = 0.997$, coefficient for the value loss $\beta = 0.25$, entropy cost of $\eta = 0.01$, and a learning rate of 0.0008. We use a standard noncentered RMSProp as a gradient ascent optimizer. The weights are initialized using orthogonal initialization. The learning updates are batched across rollouts of 6 actor steps for 16 parallel copies of the environment, totaling a mini-batch size of 96. All experiments use gradient clipping by norm to avoid exploding gradients (a familiar phenomenon with LSTMs), with a maximum norm of 0.5. Note that with 14 400 placements, we only explore 10^{-6}% of the total parameter space.

Fig. 9 depicts the RL training workflow in each environment. We start by selecting a random netlist and a random parameter set for a given environment. Each environment is reset when a stop condition is met. Training on K environments in parallel, each performing a placement on a different netlist, the reward signal is averaged on the K netlists for the network updates, which decreases the reward variance and ultimately helps the network generalize to unseen netlists as prescribed in [27] and the presented multitask methodology.

More formally, we first draw per environment k a random netlist uniformly from the set of available training netlists $n_k \sim \mathcal{U}(\mathcal{N})$. This netlist is fixed until the environment is reset. We also draw a random initial parameter setting from all possible parameter settings $p_{0,k} \sim \mathcal{U}(\mathcal{P})$. Then, while the stopping criterion is not respected, we draw $a_{j,k} \sim \pi_{\boldsymbol{\theta}}(p_{j-1,k}, n_k)$ and $p_{j,k} = a_{j,k}(p_{j-1,k})$. The pair

TABLE VI
COMPARISON OF HALF-PERIMETER BOUNDING BOX (HPWL) AFTER PLACEMENT ON TRAINING NETLISTS AMONG HUMAN DESIGN, MAB [3], AND OUR RL-BASED METHOD. HPWL IS REPORTED IN m. Δ DENOTES PERCENTAGE NEGATIVE IMPROVEMENT OVER HUMAN DESIGN

Netlist	human	MAB [3] (Δ%)	RL (Δ%)
PCI	0.010	0.0092 (−8.0%)	0.0092 (−8.0%)
DMA	0.149	0.139 (−6.7%)	0.135 (−9.4%)
B19	0.30	0.28 (−6.7%)	0.28 (−6.7%)
DES	0.42	0.37 (−11.9%)	0.36 (−14.3%)
VGA	1.52	1.40 (−7.9%)	1.41 (−7.2%)
ECG	0.72	0.65 (−9.7%)	0.68 (−5.5%)
Rocket	1.33	1.27 (−4.5%)	1.20 (−9.8%)
AES	1.49	1.44 (−2.7%)	1.40 (−6.0%)
AVC-Nova	1.59	1.49 (−6.3%)	1.46 (−8.2%)
Tate	1.53	1.42 (−7.2%)	1.45 (−5.2%)
JPEG	2.14	1.96 (−8.4%)	1.88 (−12.2%)

$(p_{j,k}, n_k)$ is used to perform placement in the EDA tool environment. The network agent policy network $\pi_{\boldsymbol{\theta}}$ is updated with a sliding history (rollout) of length J of states $\{(p_{j,k}, n_k)\}_{j\in J,k\in K}$ and corresponding rewards $\{R_{j,k}\}_{j\in J,k\in K}$ of the $k \in K$ environments, totaling a batch size of $J \times K$, using the A2C update formulas. Our reset criterion is the same for all k environments (calculated independently), so k is omitted

$$\begin{aligned} \{R_{j-4}, R_{j-3}, .., R_j\} &< 0 \| \\ \{a_{j-4}, a_{j-3}, .., a_j\} &= \text{NOTHING} \| \\ j - j_{\text{last reset}} &= 16 \quad (16\,\text{steps have passed}) \\ &\implies \text{reset}\, p_j \sim \mathcal{U}(\mathcal{P}). \end{aligned} \tag{24}$$

B. Netlist Training Results

For comparison, we use the state-of-the-art tool auto-tuner OpenTuner [3] that we adapt for Innovus as a baseline. In this framework, a MAB selects at each iteration a search technique among GA, Differential Evolution, Simulated Annealing, Torczon hillclimber, Nelder-Mead, and Particle Swarm Optimization, based on a score that forces a tradeoff between exploitation (use arm that worked best) and exploration (use a more rarely used arm). We run the search techniques in parallel, simultaneously evaluating 16 candidate parameter sets. It is run on the 11 training netlists, and we record the best-achieved wire length, performing 1300 placements per netlist so that the total number of placements equals those of the RL agent training.

Table VI shows the best wire lengths the MAB and the RL agent found during training. The human baseline is set by an experienced engineer who tunes the parameters for a day. We see that the RL agent outperforms MAB on most netlists, reducing HPWL by 9.8% on the Rocket Core benchmark. All in all, both methods improve pretty well on the human baseline.

C. Unseen Netlist Testing Results

To verify the ability of our agent to generalize, we study its performance on the LDPC netlist and the three unseen test netlists. Without additional training (the network weights are fixed), the RL agent iteratively improves a random initial parameter set by selecting action a with the highest probability $\pi_{\boldsymbol{\theta}}(a)$, as described in (10)—but ignoring the action NOTHING to avoid being stuck in the same parameter state. Because our actions are deterministic, the resulting parameters are known and directly fed back as input to the network. We repeat this process until the estimated value predicted by the

TABLE VII
COMPARISON ON LDPC AND THE THREE TEST NETLISTS OF BEST WIRE LENGTH FOUND (ONE ITERATION = ONE PLACEMENT PERFORMED). HPWL IS REPORTED IN m

Netlist	human	#iter.	MAB [3]	#iter.	RL	#iter.
LDPC	1.14	20	1.04 (−8.8%)	50	1.02 (−10.5%)	1
OpenPt	5.26	20	5.11 (−2.9%)	50	4.99 (−5.1%)	1
Netcard	4.88	20	4.45 (−8.8%)	50	4.34 (−11.1%)	1
Leon3	3.52	20	3.37 (−4.3%)	50	3.29 (−6.5%)	1

TABLE VIII
BEST PLACEMENT PARAMETERS FOUND FOR THE NETCARD BENCHMARK

Name	human	MAB [3]	RL
eco max distance	0	54	81
legalization gap	0	1	5
max density	0.90	0.92	0.94
eco priority	placed	eco	fixed
activity power driven	standard	none	none
wire length opt	medium	high	none
blockage channel (for macros)	soft	none	soft
timing effort	medium	medium	high
clock power driven	standard	none	none
congestion effort	medium	low	high
clock gate aware	true	true	true
uniform density	false	false	false

network decreases for three consecutive updates and backtrack to the settings with the highest value—this way, a "good" candidate parameter set is found without performing any placement, i.e., without interacting with the environment. We present the details for the criterion choice in the following section. We then perform a unique placement with that parameter set and record the obtained wire length.

In comparison, the MAB needs the reward signal to propose a new set of parameters and therefore needs actual placements to be performed by the tool. We track the best wire length found, allotting 50 sequential iterations to the MAB.

The best wire length found by our RL agent and the MAB on the LDPC netlist and all three test netlists is shown in Table VII. Our RL agent achieves superior wire lengths consistently, performing only one placement. Finally, Table VIII shows the best parameter settings found by the MAB and the RL agent on the Netcard benchmark. Interestingly, we can see how the two optimizers found separate ways to minimize HPWL: WL-driven versus congestion-driven.

D. Parameter Selection Criterion and Value Trajectory

We must define a criterion to select a suitable parameter set at testing time, i.e., decide when to stop the state trajectory of the state transition diagram (10) without interacting with the environment. This criterion is fundamentally distinct from the training's stopping criterion of (24), which directly interacts with the training environments to reset them by inspecting the actual rewards. Moreover, contrary to training where the entropy regularization allows exploration, the network actions are deterministic during testing. Therefore, it is not evident that the testing criterion can be best defined by simply replacing the reward with the predicted value in the first line of (24). Indeed, we observed that our trained network did not achieve superior performance to the MAB if used in this fashion.

Therefore, after first fixing the pretrained weights of the network, we derive a generic selection criterion on a unique held-out unseen netlist (LDPC, the smallest test benchmark) by tracking

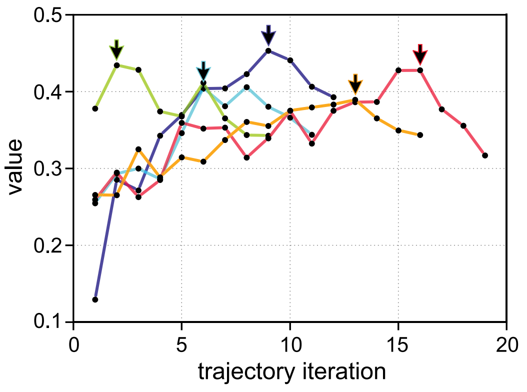

Fig. 10. Value function trajectories on the Leon3 benchmark for five random initial parameter sets. Arrows indicate the highest value on a given trajectory.

TABLE IX
PPA COMPARISON AFTER ROUTING ON THE SELECTION CRITERION DEFINITION (LDPC) AND TEST SETS. THE TARGET FREQUENCIES ARE 1 GHZ, 500 MHZ, 833 MHZ, AND 666 MHZ FROM TOP TO BOTTOM

Netlist	Metric	human	MAB [3]	RL
LDPC	WL (m)	1.65	1.57	1.53
	WNS (ns)	−0.005	−0.001	−0.001
	Power (mW)	162.10	156.49	153.77
OpenPiton	WL (m)	6.31	6.24	6.10
	WNS (ns)	−0.003	−0.001	0
	Power (mW)	192.08	190.95	189.72
NETCARD	WL (m)	8.01	7.44	7.15
	WNS (ns)	−0.006	−0.007	−0.004
	Power (mW)	174.05	170.51	167.70
LEON3	WL (m)	5.66	5.53	5.41
	WNS (ns)	−0.005	−0.001	−0.003
	Power (mW)	156.83	156.00	155.51

the trajectories of the network's predicted value and environment reward feedback. Empirically, the found criterion yields good HPWL improvements on all test netlists despite being derived from LDPC only. In Fig. 10, we show the trajectories of the value function on benchmark Leon3 for multiple random starting points. The randomness of the starting parameter set does not notably affect the convergence of the value function.

E. PPA Comparison After Routing

To confirm the improvement in HPWL after placement is translated into one in the final routed wire length, we perform the routing of the placed designs. They are all routed with the same settings where metal layers 1–6 are used. The layouts of OpenPiton Core are shown in Fig. 11. We verify the target frequency is achieved and routing succeeded without congestion issues or DRC violations. The PPA of routed designs is summarized in Table IX. We observe that the HPWL reduction after placement is conserved after routing on all test designs, reaching 7.3% and 11% wire length savings on LDPC and Netcard compared with the human baseline. Footprints are 74 283 μm^2 for LDPC, 1 199 934 μm^2 for OpenPiton, 728 871 μm^2 for Netcard, and 894 115 μm^2 for Leon3.

F. Controlled Feature Selection

One might wonder if the proposed handcrafted and GNN graph features help our RL agent's superior performance. We resort to well-established controlled variable selection methods to offer evidence of their importance. The RL agent's strength is closely

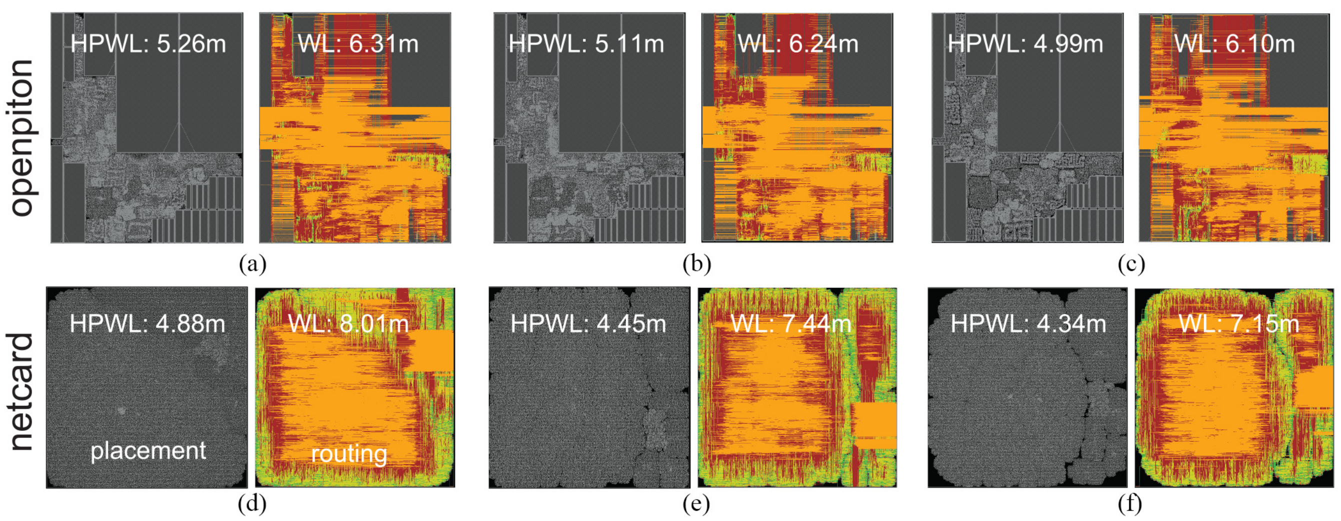

Fig. 11. 28-nm full-chip GDSII layouts of OpenPiton and Netcard. Top: (a) human design (took 7 h). (b) MAB (took 16 h). (c) RL (took 20 min). Bottom: (a) human design (took 8 h). (b) MAB (took 20 h). (c) RL (took 25 min).

related to its optimal action prediction and value function estimation. Therefore, we study which explanatory variables serve these two agent predictions.

We thus store the training tuples $(s_t, a_t, v(s_t))$ to examine the relationships between state and corresponding action and value. We only keep the samples of iterations ≥ 100, resulting in 4800 tuples, which by Fig. 8 correspond to a network of good quality that has converged.

Let $y \in \{a, v\}$ denote the response of interest, the action, or value, respectively, which potentially depends on all state variables $X_1, \ldots, X_p$. To discover the importance of variables, we assume an (explainable) linear model

$$y = X \cdot \beta + z, \text{ where } z_i \overset{\text{i.i.d.}}{\sim} \mathcal{N}(0, \sigma^2). \tag{25}$$

We have $n = 4800$ observations, $p = 64$ features, and β is the unknown vector of coefficients and is supposed to be sparse. We wish to select a set of features X_j that are likely to be relevant to the response y, without too many false positives. The most pertinent way to measure the performance of the discovery method is the false discovery rate (FDR)

$$\text{FDR} = \mathbb{E}\left[\frac{\text{\# false positives}}{\text{\# of features selected}}\right] = \mathbb{E}\left[\frac{|\hat{S} \cap H_0|}{\max(1, |\hat{S}|)}\right] \tag{26}$$

where $\hat{S}$ is the set of selected features and H_0 = "null hypotheses" $= \{j \in [\![1..p]\!] : \beta_j^\star = 0\}$, where $\beta_j^\star$ denotes the "true" value of β_j from nature. This is the frame of multiple hypothesis testing, where rejecting the null hypothesis $\{\beta_j = 0\}$ indicates X_j is a likely explanatory variable.

We propose two methods for this large-scale hypothesis testing problem. Using two methods built on different sets of theoretical assumptions will allow us to derive reliable conclusions when they agree.

1) We first run the Benjamini–Hochberg procedure (BHq) [28] on a linear regression coefficient estimate. The hypotheses are first ordered and then rejected or accepted based on their p-values $(= \mathbb{P}[|t| \geq |t_j| \mid H_0])$ obtained from a two-sample student's t-test $(t_j = \hat{\beta}_j / \operatorname{sd}(\hat{\beta}_j))$.
2) We also run the Knockoff [29] with the Lasso. This method manufactures fake variables mimicking the original variables' correlation structure. These negative controls allow identifying the significant predictors while controlling the FDR. The test statistic is $W_j = \max\{\lambda_j, \tilde{\lambda_j}\} \cdot \operatorname{sign}(\lambda_j - \tilde{\lambda_j})$, where λ_j and $\tilde{\lambda_j}$ are the first time the jth variable and its knockoff enter the Lasso path, respectively. Then, the variables with W_j over a certain threshold computed from the target FDR are selected.

Both methods return the set of nonzero β_j, on which the null hypothesis was rejected. We use a traditional target of FDR $\leq q = 0.1$. The features selected by the two testing methodologies are presented in Tables X and XI, listed in decreasing order of importance.

TABLE X
CONTROLLED FEATURE SELECTION FOR THE VALUE ESTIMATION OF OUR TRAINED RL AGENT

Knockoffs [29]		BHq [28]	
Name	W_j	Name	$\lvert t\rvert$-statistic
# IO	17.10	# IO	75.02
GNN(3)	3.97	# FF	23.20
Fiedler value	3.04	$\overline{CC}$	21.18
total cell area	2.88	# nets w. fan. $\in]5, 10[$	17.31
GNN(9)	2.72	max density	16.57
# nets w. fan. $\in]5, 10[$	1.68	Fiedler value	6.36
max clique	1.58	total cell area	6.16
max density	1.09	GNN(16)	3.69
GNN(24)	0.57	-	-

TABLE XI
CONTROLLED FEATURE SELECTION FOR THE ACTION PREDICTION OF OUR TRAINED RL AGENT

Knockoffs [29]		BHq [28]	
Name	W_j	Name	$\lvert t\rvert$-statistic
clock gate aware	6.30	uniform density	4.72
congestion effort	4.18	max density	4.37
legalization gap	2.39	legalization gap	4.14
max density	2.09	# nets w. fan. $\in]5, 10[$	4.11
largest SCC	1.92	wire length opt	3.90
Fiedler value	0.73	clock gate aware	2.19
-	-	timing effort	1.91
-	-	# IO	1.87

The high values of test statistics $|t|$ and W show strong evidence that the graph features are essential to the RL value network. Interestingly, the number of IOs is deemed most important. This is not surprising, as combined with the total cell area, it closely relates to the famous Rent's rule, which is empirically seen as a reasonable estimate of a circuit wire length [30]. Overall, we observe a good balance of GNN and handcrafted graph features in the value prediction. In contrast, tool parameters emerge as more critical for the action prediction, while a few graph features still appear.

VI. Statistical Testing

The above experiments show that our RL agent outperforms a human designer and the MAB [3] on some placement parameter tuning task instances. However, we recognize that the various sources of randomness in these test experiments, such as the starting parameters and our handcrafted stopping and selection rules, may introduce a selection bias to the results. Hence, we present a formal method to prove the statistical significance of the superiority of our RL agent in this section.

A. Goal

In most RL works, the strength of an agent is established by a win/loss percentage over a human or competing agent. This procedure applies to tasks that can be simulated quickly and are suited for benchmarking computer players in games, such as Go, Chess, and Atari Games. However, the latency associated with EDA tool runs, sometimes up to hours or even days, makes theoretical testing of such strength metric prohibitive, as many experimental samples need to be collected to prove or disprove it with high confidence. In addition, the intrinsic randomness of EDA algorithms that rely on heuristics heavily is also a challenge in assuring a particular method works well, especially when accounting for the diversity of possible netlists.

For this reason, we turn toward sequential testing, where data is collected and evaluated sequentially, and decisions are made as soon as guaranteed by a stopping rule with a theoretical basis. This approach decreases the cost of additional sampling.

B. Problem Definition

Similar to the testing procedure presented in the previous section, the RL agent and MAB have already been trained. We then make them perform against each other where each session follows the method of Section V-C. Each session, referred to as a *game*, has a binary outcome defined from the floating-point value of HPWL as

$$\begin{aligned}&\text{RL agent wins if } \ \text{HPWL}_{\text{agent}} < \text{HPWL}_{\text{MAB}}\\&\text{RL agent loses otherwise.}\end{aligned} \tag{27}$$

The games are repeated one after another. We consider their outcomes independent as the RL agent starts with a random parameter set before each game. The netlist is, on the other hand, fixed.

This description naturally fits the Bernoulli distribution Bernoulli(θ) of the coin-toss random variable by setting

$$\begin{aligned}X_i &= 1 \quad \text{if RL agent wins game}\, i,\\X_i &= 0 \quad \text{if RLagent loses game}\, i.\end{aligned} \tag{28}$$

Each game is associated with a random variable X_i encoding the result of the game i, leading to i.i.d. random variables $X_1, \dots, X_n$ after n games. Moreover, each X_i follows a distribution Bernoulli(θ) for $\theta \in [0, 1]$, where θ corresponds to our RL agent's probability of winning a game, namely, $\mathbb{P}[X_i = 1]$.

We wish to discover meaningful information on θ, where a value above 0.5 corresponds to the superiority of our agent. However, the exact value of θ is only available asymptotically, i.e., as a win/loss percentage of an infinite amount of games.

Traditional methods offer approximations of θ that are accurate for large sample sizes only. Namely, consider the maximum-likelihood estimator (MLE) of θ, of the form

$$\hat{\theta}_n = \frac{1}{n}\sum_{i=1}^{n} X_i. \tag{29}$$

A confidence interval is introduced to include uncertainty in this estimation, where we know that θ is in the interval with a probability larger than a certain percentage. Only nonasymptotic coverage methods, where the coverage properties hold for any value of the number of games played, are acceptable in our case, as asymptotic methods would rely on a prohibitive number of games. A standard tight confidence interval at level $1 - \alpha$ of the Bernoulli model is $[\hat{\theta}_n \pm \sqrt{\log(2/\alpha)/(2n)}]$, which means the inequality

$$\mathbb{P}\left[\hat{\theta}_n - \sqrt{\frac{\log(2/\alpha)}{2n}} \le \theta \le \hat{\theta}_n + \sqrt{\frac{\log(2/\alpha)}{2n}}\right] \ge 1 - \alpha \tag{30}$$

is satisfied. Per this formula, if we were to play 50 games and desire a 99% ($\alpha = 1\%$) confidence in the value of our estimation, the uncertainty would be ± 0.23, which can be substantial depending on the value of $\hat{\theta}_n$.

Thus, rather than using the estimation presented above, we propose a statistical experiment with data X and model $\{\text{Bernoulli}(\theta) : \theta \in [0, 1]\}$. We want to decide between the composite null hypothesis $\{H_0 : \theta \le \theta_0\}$ against the composite alternative $\{H_1 : \theta \ge \theta_1\}$ where $\theta_1 > \theta_0$. We decide that H_0 means that our RL agent is not better, while H_1 means that our agent is indeed better. The values of θ_0 and θ_1 can be chosen to fit the definition of superiority.

When testing hypotheses, two types of errors are considered, namely, Type-I error and Type-II error, defined as

$$\begin{aligned}\alpha &= \mathbb{P}_\theta[\text{reject} \quad H_0] \quad \text{when}\, \theta \le \theta_0,\\\beta &= \mathbb{P}_\theta[\text{reject} \quad H_1] \quad \text{when}\, \theta \ge \theta_1.\end{aligned} \tag{31}$$

In our problem, the Type I error corresponds to the probability of our RL agent falsely being evaluated as better. The Type II error corresponds to the probability of falsely evaluating the agent as inferior. Ideally, we want the Type I and Type II errors to be minor. In practice, H_0 and H_1 must be chosen to make the corresponding Type I error more severe than Type II. This fact motivated our choice of H_0: we stay conservative and emphasize avoiding evaluating our agent incorrectly as better.

C. Sequential Answer

Wald [31] proposed a general procedure called the SPRT that can test our two competing hypotheses with a small number of samples. The procedure evaluates the likelihood ratio between the hypotheses and stops acquiring observations when this ratio crosses some boundary.

After each game, Wald's SPRT methodology computes the likelihood ratio L_n of the currently n played games, i.i.d. observations $X_1, \dots, X_n$ having common density function $P \in \{P_0, P_1\}$, the competing hypotheses

$$L_n = \prod_{i=1}^{n} P_1(X_i)/P_0(X_i). \tag{32}$$

The SPRT stops sampling at stage

$$N = \inf\{n \geq 1 : L_n \notin (A, B)\} \tag{33}$$

where $0 < A < 1 < B$ are stopping boundaries, rejecting $\{H_0 : P = P_0\}$, and accepting $\{H_1 : P = P_1\}$ if $L_n \geq B$. Until the stopping boundaries are not crossed, we must play more games. The choice of A and B is dictated by the Type I and Type II error probabilities α and β, respectively. Wald has shown that α and β are related to A and B by the inequalities

$$\alpha \leq A^{-1}(1-\beta), \quad \beta \leq B(1-\alpha) \tag{34}$$

which provides approximate determinations (nominal error rates) of A and B in terms of the error probabilities.

The SPRT has minimal stopping time $\mathbb{E}_\theta[T]$ at $\theta = \theta_0$ or θ_1, where T denotes the sample size, but its $\mathbb{E}_\theta[T]$ for different θ may not be optimal [32]. Its correctness, however, stands, and to test $\{H_0 : \theta \leq \theta_0\}$ against $\{H_1 : \theta \geq \theta_1\}$ with Type I and Type II error probabilities not exceeding α and β, one can use the SPRT of $\{J : \theta = \theta_0\}$ versus $\{K : \theta = \theta_1\}$ with Type I and Type II error probabilities α and β.

D. SPRT in Our Problem

For simplicity, we present the SPRT stopping conditions by considering first the exponential family of distributions, $\mathcal{G} = \{g_\eta(x) = e^{\eta x - \psi(\eta)} g_0(x),\ \eta \in \mathcal{A},\ x \in \mathcal{X}\}$, where $\mathcal{A}$ and $\mathcal{X}$ are subsets of the real line $\mathcal{R}^1$. η is the canonical parameter, and x is the natural statistic; in many cases, the observed data point. For the exponential family, it is straightforward to show that $L_n \in (A, B)$ is equivalent to

$$\frac{\log(A) + n(\psi(\eta_1) - \psi(\eta_0))}{\eta_1 - \eta_0} \leq \sum_{i=1}^{n} x_i \leq \frac{\log(B) + n(\psi(\eta_1) - \psi(\eta_0))}{\eta_1 - \eta_0}. \tag{35}$$

The Bernoulli family, where $\mathbb{P}_\theta[X = x] = \theta^x (1-\theta)^{1-x}$, can be written in exponential form with the correspondences

$$\begin{cases} \eta = \log(\frac{\theta}{1-\theta}) & (\theta = \frac{1}{1+e^{-\eta}}) \\ \psi(\eta) = -\log(1-\theta) & (= \log(1+e^{\eta})) \\ g_0(x) = 1 \end{cases} \tag{36}$$

yielding for our two hypotheses H_0 against H_1

$$\eta_1 - \eta_0 = \log\left(\frac{\theta_1(1-\theta_0)}{\theta_0(1-\theta_1)}\right)$$
$$\psi(\eta_1) - \psi(\eta_0) = \log\left(\frac{1-\theta_0}{1-\theta_1}\right) \tag{37}$$

which can be directly used in (35), along with n as the current number of games played and $\sum_{i=1}^{n} x_i$ as the number of wins accumulated by our RL agent during these n games.

E. Experiments

1) Conditions: We calculate the stopping boundaries A and B for a small error rate target of $\alpha = \beta = 1\%$ using (34) to provide a very accurate decision. Moreover, to be conservative, we pose $\{H_0 : \theta \leq \theta_0 = 0.5\}$ and $\{H_1 : \theta \geq \theta_1 = 0.8\}$. Setting θ_1 as high permits a "higher superiority" level tolerance if our agent is found better by the statistical procedure.

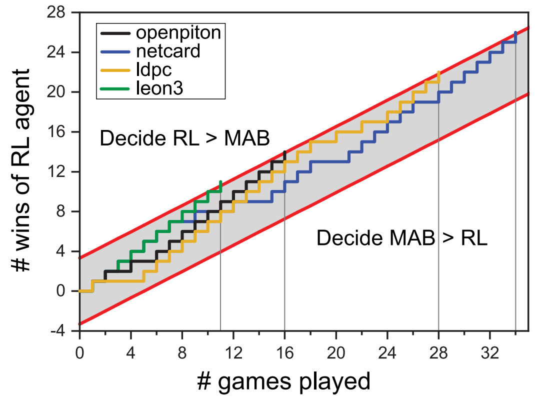

Fig. 12. SPRT [31] of the opposition of MAB [3] versus our trained RL agent. The latter is "superior" to the MAB on all four LDPC and unseen test netlists.

2) Results: We plot in Fig. 12 the results of the games of our RL agent against the MAB for the LDPC and the three test netlists. The red lines correspond to the bounds of (35). For all netlists, the SPRT leads to the acceptance of the hypothesis H_1, indicating the superiority of our RL agent at the parameter tuning task. Noticeably, the RL agent is considerably stronger on the Leon3 and OpenPiton benchmarks, as it is declared better much faster than in the LDPC and Netcard cases.

On the other hand, consider the MLE and its associated nonasymptotic coverage in (30). The stopping times of the SPRT procedure in Fig. 12 would yield the realization of the MLE confidence intervals at level 99% as [0.47, 1] for OpenPiton ($n = 16$), [0.49, 1] for Netcard ($n = 34$), [0.48, 1] for LDPC ($n = 28$), and [0.51, 1] for Leon3 ($n = 11$). Except for Leon3, these intervals intersect with $[0, \theta_0 = 0.5]$, allowing for H_0 to be true—which is rather quickly disproved by the SPRT. Regardless, in all cases, they are never contained in $[\theta_1 = 0.8, 1]$ and thus can never prove H_1. Therefore, the proposed sequential approach is superior in offering an adequate testing methodology of the placement task's performance with a smaller sample size.

VII. Conclusion

Our placement parameter optimizer based on deep RL provides a preset of improved parameter settings without human intervention. This is crucial to shift from the CAD expert-user mindset to a Design Automation mindset. We use a novel representation to formulate states and actions applied to placement optimization. Our experimental results show that our agent generalizes well to unseen netlists and consistently reduces wire length compared with a state-of-the-art tool auto-tuner in only one iteration without additional training. Furthermore, we demonstrate the superiority of our agent using an SPRT.

References

[1] D. Silver *et al.*, "Mastering chess and Shogi by self-play with a general reinforcement learning algorithm," 2017, *arXiv:1712.01815*.

[2] I. L. Markov, J. Hu, and M.-C. Kim, "Progress and challenges in VLSI placement research," *Proc. IEEE*, vol. 103, no. 11, pp. 1985–2003, Nov. 2015.

[3] J. Ansel *et al.*, "OpenTuner: An extensible framework for program autotuning," in *Proc. 23rd Int. Conf. Parallel Archit. Compilation (PACT)*, 2014, pp. 303–315.

[4] E. Ustun, S. Xiang, J. Gui, C. Yu, and Z. Zhang, "LAMDA: Learning-assisted multi-stage autotuning for FPGA design closure," in *Proc. IEEE 27th Annu. Int. Symp. Field-Programmable Custom Comput. Mach. (FCCM)*, 2019, pp. 74–77.

[5] B. Shahriari, K. Swersky, Z. Wang, R. P. Adams, and N. De Freitas, "Taking the human out of the loop: A review of Bayesian optimization," *Proc. IEEE*, vol. 104, no. 1, pp. 148–175, Jan. 2016.

[6] M. M. Ziegler, H.-Y. Liu, G. Gristede, B. Owens, R. Nigaglioni, and L. P. Carloni, "A synthesis-parameter tuning system for autonomous design-space exploration," in *Proc. Design Autom. Test Europe Conf. Exhibit. (DATE)*, 2016, pp. 1148–1151.

[7] N. Kapre, B. Chandrashekaran, H. Ng, and K. Teo, "Driving timing convergence of FPGA designs through machine learning and cloud computing," in *Proc. IEEE 23rd Annu. Int. Symp. Field-Programmable Custom Comput. Mach.*, 2015, pp. 119–126.

[8] Z. Xie *et al.*, "FIST: A feature-importance sampling and tree-based method for automatic design flow parameter tuning," in *Proc. 25th Asia South Pacific Design Autom. Conf. (ASP-DAC)*, 2020, pp. 19–25.

[9] J. Kwon, M. M. Ziegler, and L. P. Carloni, "A learning-based recommender system for autotuning design flows of industrial high-performance processors," in *Proc. 56th Annu. Design Autom. Conf.*, 2019, pp. 1–6.

[10] T. Chen *et al.*, "Learning to optimize: A primer and a benchmark," 2021, *arXiv:2103.12828*.

[11] Y.-W. Chang, "Circuit placement challenges: Technical perspective," *Commun. ACM*, vol. 56, no. 6, p. 104, 2013.

[12] A. B. Kahng, "Advancing placement," in *Proc. Int. Symp. Phys. Design*, 2021, pp. 15–22.

[13] R. E. Tarjan, "Depth-first search and linear graph algorithms," *SIAM J. Comput.*, vol. 1, no. 2, pp. 146–160, 1972.

[14] C. Bron and J. Kerbosch, "Algorithm 457: Finding all cliques of an undirected graph," *Commun. ACM*, vol. 16, no. 9, pp. 575–577, 1973.

[15] O. Coudert, "Exact coloring of real-life graphs is easy," in *Proc. 34th Design Autom. Conf.*, 1997, pp. 1–6.

[16] V. Colizza, A. Flammini, M. A. Serrano, and A. Vespignani, "Detecting rich-club ordering in complex networks," *Nat. Phys.*, vol. 2, pp. 110–115, Jan. 2006.

[17] D. J. Watts and S. H. Strogatz, "Collective dynamics of 'small-world' networks," *Nature*, vol. 393, pp. 440–442, Jun. 1998.

[18] R. B. Lehoucq and D. C. Sorensen, "Deflation techniques for an implicitly restarted Arnoldi iteration," *SIAM J. Matrix Anal. Appl.*, vol. 17, no. 4, pp. 789–821, 1996.

[19] W. Hamilton, Z. Ying, and J. Leskovec, "Inductive representation learning on large graphs," in *Advances in Neural Information Processing Systems*, I. Guyon *et al.*, Eds. Red Hook, NY, USA: Curran Assoc., Inc., 2017.

[20] L. van der Maaten and G. Hinton, "Visualizing data using t-SNE," *J. Mach. Learn. Res.*, vol. 9, no. 86, pp. 2579–2605, 2008.

[21] V. Mnih *et al.*, "Asynchronous methods for deep reinforcement learning," in *Proc. 33rd Int. Conf. Mach. Learn.*, 2016, pp. 1928–1937.

[22] L. Espeholt *et al.*, "IMPALA: Scalable distributed deep-RL with importance weighted actor-learner architectures," in *Proc. 35th Int. Conf. Mach. Learn.*, 2018, pp. 1407–1416.

[23] O. Vinyals *et al.*, "StarCraft II: A new challenge for reinforcement learning," 2017, *arXiv:1708.04782*.

[24] M. Luong, H. Pham, and C. D. Manning, "Effective approaches to attention-based neural machine translation," 2015, *arXiv:1508.04025*.

[25] Y. Zhang and Q. Yang, "A survey on multi-task learning," *IEEE Trans. Knowl. Data Eng.*, early access, Mar. 31, 2021, doi: 10.1109/TKDE.2021.3070203.

[26] G. Brockman *et al.*, "OpenAI gym," 2016, *arXiv:1606.01540*.

[27] A. Mirhoseini *et al.*, "Chip placement with deep reinforcement learning," 2020, *arXiv:2004.10746*.

[28] Y. Benjamini and Y. Hochberg, "Controlling the false discovery rate: A practical and powerful approach to multiple testing," *J. Royal stat. Soc. B, Methodol.*, vol. 57, no. 1, pp. 289–300, 1995.

[29] R. F. Barber and E. J. Candès, "Controlling the false discovery rate via knockoffs," *Ann. Stat.*, vol. 43, no. 5, pp. 2055–2085, Oct. 2015.

[30] W. Donath, "Placement and average interconnection lengths of computer logic," *IEEE Trans. Circuits Syst.*, vol. 26, no. 4, pp. 272–277, Apr. 1979.

[31] A. Wald, "Sequential tests of statistical hypotheses," *Ann. Math. Stat.*, vol. 16, no. 2, pp. 117–186, 1945.

[32] T. L. Lai, "Likelihood ratio identities and their applications to sequential analysis," *Sequential Anal.*, vol. 23, no. 4, pp. 467–497, 2004.

Anthony Agnesina received the Diplôme d'Ingénieur degree from CentraleSupélec, Gif-sur-Yvette, France, in 2016, and the M.S. and Ph.D. degrees in electrical and computer engineering from Georgia Tech, Atlanta, GA, USA, in 2017 and 2022, respectively.

He joined NVIDIA Research, Austin, TX, USA, in 2022. His research interests include 3-D integrated circuits, applied machine learning to EDA, and algorithms for computer-aided design of VLSI circuits.

Kyungwook Chang (Member, IEEE) received the B.S. and M.S. degrees in electrical and computer engineering from Seoul National University, Seoul, South Korea, in 2007 and 2010, respectively, and the Ph.D. degree from the School of Electrical and Computer Engineering, Georgia Institute of Technology, Atlanta, GA, USA, in 2019.

He joined the College of Information and Communication Engineering, Sungkyunkwan University, Suwon, South Korea, in 2021, where he serves as an Assistant Professor. His current research interests include computer-aided design solutions for 3-D ICs, design-technology co-optimization, physical/logic design and analysis, power delivery network, and parallel computing/memory architecture.

Sung Kyu Lim (Senior Member, IEEE) received the B.S., M.S., and Ph.D. degrees from the Computer Science Department, University of California at Los Angeles, Los Angeles, CA, USA, in 1994, 1997, and 2000, respectively.

In 2001, he joined the School of Electrical and Computer Engineering, Georgia Institute of Technology, Atlanta, GA, USA, where he is a Motorola Solutions Foundation Professor. His research is featured as Research Highlight in the Communication of the ACM in January 2014. He is the author of *Practical Problems in VLSI Physical Design Automation* (Springer, 2008). He has published more than 400 papers on 2.5-D and 3-D ICs. His research focus is on the architecture, design, test, and electronic design automation solutions for 2.5-D and 3-D ICs.

Dr. Lim received the National Science Foundation Faculty Early Career Development (CAREER) Award in 2006. He received the ACM SIGDA Distinguished Service Award in 2008. He received the Best Paper Award from the IEEE TRANSACTIONS ON ELECTROMAGNETIC COMPATIBILITY and the IEEE TRANSACTIONS ON COMPUTER-AIDED DESIGN OF INTEGRATED CIRCUITS AND SYSTEMS in 2022. He was an Associate Editor of the IEEE TRANSACTIONS ON VERY LARGE SCALE INTEGRATION (VLSI) SYSTEMS from 2007 to 2009 and the IEEE TRANSACTIONS ON COMPUTER-AIDED DESIGN OF INTEGRATED CIRCUITS AND SYSTEMS from 2013 to 2018.