VLSI Placement Parameter Optimization using Deep Reinforcement Learning

Anthony Agnesina, Kyungwook Chang, and Sung Kyu Lim
School of ECE, Georgia Institute of Technology, Atlanta, GA
agnesina@gatech.edu

ABSTRACT

The quality of placement is essential in the physical design flow. To achieve PPA goals, a human engineer typically spends a considerable amount of time tuning the multiple settings of a commercial placer (e.g. maximum density, congestion effort, etc.). This paper proposes a deep reinforcement learning (RL) framework to optimize the placement parameters of a commercial EDA tool. We build an autonomous agent that learns to tune parameters optimally 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 along with graph embeddings generated using unsupervised Graph Neural Networks. Our RL algorithms are chosen to overcome the sparsity of data and latency of placement runs. Our trained RL agent achieves up to 11% and 2.5% wirelength 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× less iterations).

1 INTRODUCTION

In the recent years, the combination of deep learning techniques with reinforcement learning (RL) principles has resulted in the creation of self-learning agents achieving superhuman performance at the game of Go, Shogi and Chess [13]. Deep RL is also used with large success in real-world applications such as robotics, finance, self-driving cars, etc.

The quality of VLSI placement is essential for the subsequent steps of physical design with influential repercussions on design quality and design closure. Recent studies [10] however 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 state-of-the-art placers is to minimize the total interconnect length, i.e. the estimated half-perimeter wire length (HPWL) from the placed cells locations.

This material is based upon work supported by the National Science Foundation under Grant No. CNS 16-24731 and the industry members of the Center for Advanced Electronics in Machine Learning.

ICCAD '20, November 2–5, 2020, Virtual Event, USA

ACM ISBN 978-1-4503-8026-3/20/11…$15.00
https://doi.org/10.1145/3400302.3415690

The algorithms implemented inside the 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 on tuning and running a commercial placer than on creating a first version of the design. Tools and flows have steadily increased in complexity, with modern place and route tools offering more than 10,000 parameter settings. Expert users are required in particular for the latest technology nodes, with increased cost and risk. Indeed, as the design space of the parameters is too big 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 metaheurisitics. Moreover, the advertised goal of a parameter may not always directly translate onto the targeted metric.

A state-of-the-art tool auto-tuner [1] is used in EDA such as in [18] and [19] to optimize Quality of Results (QoR) in the FPGA and high-level synthesis (HLS) compilation flows. It leverages Multi-Armed 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 embeddings. This helps generalize the tuning process from netlist to netlist, saving long and costly placement iterations.

The goal of our RL framework is to learn an optimization process that finds placement parameters minimizing wirelength after placement. The main contributions of this paper are:

- We reduce the significant time expense of VLSI development by application of deep RL to pre-set the placement parameters of a commercial EDA tool. To the best of our knowledge, this is the first work on RL applied to placement parameters optimization.
- Our RL algorithm overcomes the sparsity of data and the latency of design tool runs using multiple environments collecting experiences in parallel.
- We use a mixture of features relative to graph topological characteristics along with graph embedding generated by a graph neural network to train an RL agent capable of generalizing its tuning process to unseen netlists.
- We build an autonomous agent that iteratively learns to tune parameter settings to optimize placement, without supervised samples. We achieve better wirelengths on unseen netlists than a state-of-the-art auto-tuner, without any additional training and in just one placement iteration.

2 RL ENVIRONMENT

2.1 Overview

We build an RL agent that tunes the parameter settings of the placement tool autonomously, with the objective of minimizing wirelength. Our RL problem consists of the following four key elements:

- States: the set of all netlists in the world 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.
- Actions: set of actions that the agent can use to modify the current parameters. An action a changes the setting of a subset of parameters.
- State transition: given a state (s_t) and an action, the next state (s_{t+1}) is the same netlist with updated parameters.
- Reward: minus the HPWL output from the commercial EDA placement tool. The reward increases if the action improved the parameter settings in terms of minimizing wirelength.

As depicted in Figure 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:

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

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:

$$v_\pi(s_t) = \mathbb{E}_\pi[G_t|s_t] = \mathbb{E}_\pi\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1}|s_t\right]. \tag{2}$$

2.2 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 wirelength. (5) Select a discount factor $\gamma < 1$ to force the agent reduce wirelength in as few steps as possible.

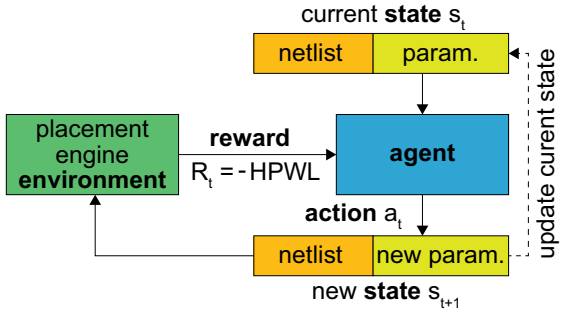

Figure 1: Reinforcement learning agent-environment interaction in the proposed methodology.

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

2.3 Our States

We define our state as the joint values of 12 placement parameters from *Cadence Innovus* used to perform the current placement (Table 1), along with information metrics on the netlist being placed. The netlist information consists of a mixture of metadata knowledge (number of cells, floorplan area, etc.) with graph topological features (Table 2) along with unsupervised features extracted using a graph neural network. Netlist characteristics are important to transfer the knowledge accross very different netlists so that our agent generalizes its tuning process to unseen netlists. Indeed, the optimal policy is likely to be related to the particularities of each netlist. 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.

2.3.1 Topological Graph Features. In order to learn rich graph representations that uniquely classify each graph, we borrow concepts from graph theory. We use Boost and BLAS optimized C++ libraries to extract efficiently our features on large graphs (all collected in less than 1hr for the larger netlists). Let $G = (V, E)$ be the directed cyclic graph representing the netlist, obtained using a fully-connected clique 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 into one and self-loops are eliminated. We consider the following graph features to capture complex topology characteristics (e.g. connections and spectral) from the netlist:

- *Strongly Connected Components (SCC):* 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 depth-first search with an algorithm due to Tarjan [14].
- *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 [3] to find the maximal cliques.
- *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

Table 1: 12 placement parameters we are targeting. The solution space is $6\mathbf{x}10^9$.

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 wirelength 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 2: 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 (um^2)	integer	$\overline{CC}$	float
# hardmacros	integer	Fiedler value	float
macro area (um^2)	integer	spectral radius	float

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 [5].

- *Logic Levels:* What is the maximum distance (# gates traversed) between two flip-flops? $LL = \max_{a,b \in FFs} d(a, b)$.
- *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 $RCC_k = \frac{2|E_k|}{|V_k|(|V_k|-1)}$ [4].
- *Clustering coefficient:* A measure of the cliquishness of nodes neighborhoods [17]:

$$\overline{CC} = \frac{1}{|V|} \sum_{i \in V} \frac{|e_{jk} : v_j, v_k \in \text{Neighbors}(i), e_{jk} \in E|}{\deg(i)(\deg(i) - 1)}. \quad (3)$$

- *Spectral characteristics:* Using the implicitly restarted Arnoldi method [8], 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 important information about the netlist. For example, connectivity features such as SCC, maximal clique and RCC are important to capture congestion considerations (considered during placement refinement) while logic levels translate indirectly the difficulty of meeting timing by extracting the longest logic path.

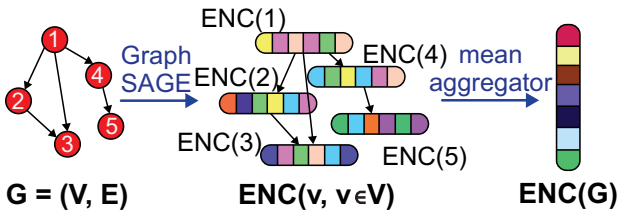

Figure 2: Graph embedding using a graph neural network package GraphSAGE. In our experiments, we first extract 32 features for each node in the graph. Next, we calculate the mean among all nodes for each feature. In the end, we obtain 32 features for the entire graph.

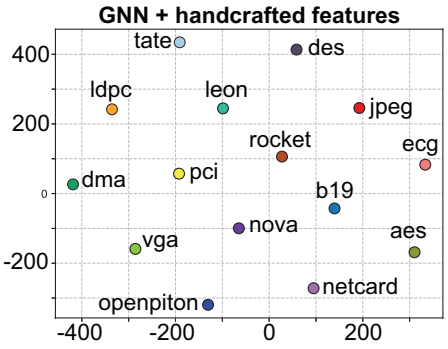

Figure 3: t-SNE visualization of our 20 handcrafted plus 32 GNN features combined. Points representative of each netlist are well separated, proving graph features capture the differences between netlists well.

2.3.2 Features from Graph Neural Network (GNN). Starting from simple node features including degree, fanout, area and encoded gate type, we generate node embeddings ($\text{ENC}(v)$) using unsupervised *GraphSAGE* [7] 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 Figure 2:

$$\text{ENC}(G) = \text{MEAN}\big(\text{ENC}(v) | v \in V\big). \quad (4)$$

Table 3: 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

The t-SNE projection in two dimensions [15] of the vector of graph features is displayed in Figure 3. We see that all netlists points are far apart, indicating that the combination of our handcrafted and learned graph features distinguish well the particularities of each netlist.

2.4 Our Actions

We 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

$$s_0 \xrightarrow{\pi(s_0)} s_1 \xrightarrow{\pi(s_1)} \dots \xrightarrow{\pi(s_{n-1})} s_n \tag{5}$$

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

Defining only two actions per placement parameter would result in 24 different actions, which is too many for the agent to learn well. Thus, we decide to first group tuning variables per type (Boolean, Enumerate, Numeric) and per placement "focus" (Global, Detailed, Effort). On each of these groups, we define simple yet expressive actions such as FLIP (for booleans), UP, DOWN, etc. For integers, we define prepared ranges where UP ≡ "put in upper range", while for enumerates DOWN ≡ "pass from high to medium" for example. We also add one arm that does not modify the current set of parameters. It serves as a trigger to reset the environment, in case it gets picked multiple times in a row. This leads to the 11 different actions $\mathcal{A}$ presented in Table 3. Our action space is designed to be as simple as possible in order to help neural network training, but also expressive enough so that any parameter settings can be reached by such transformations.

2.5 Our Reward Structure

In order to learn with a single RL agent across various netlists with different wirelengths, we cannot define a reward directly linear with HPWL. Thus, to help convergence, we adopt a normalized reward function which renders the magnitude of the value approximations similar among netlists:

$$R_t := \frac{HPWL_{\text{Human Baseline}} - HPWL_t}{HPWL_{\text{Human Baseline}}}. \tag{6}$$

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

2.6 Extensions in Physical Design Flow

The environment description and in particular the action definition can be applied to Parameter Optimization of any stage in the physical design flow, such as routing, clock tree synthesis, etc. As our actions act on abstracted representations of tool parameters, we can perform rapid design space exploration. The reward function can be easily adjusted to target PPA such as routed wirelength or congestion, in order to optimize the design for different trade-offs. For example, a reward function combining various attributes into a single numerical value can be:

$$R = \exp\left(\sum_k \alpha_k \text{QoR}_k\right) - 1. \tag{7}$$

3 RL PLACEMENT AGENT

3.1 Overview

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

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

3.2 Goal of Learning

From the many ways of learning how to behave in an environment, we choose to use what is called policy-based reinforcement learning. 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.

The goal of this optimization problem is to learn directly 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:

- It is model-free which is important as the placer tool environment is very complex and may be hard to model.
- Our intuition is that the optimal policy may be simple to learn and represent (e.g. keep increasing the effort) while the value of

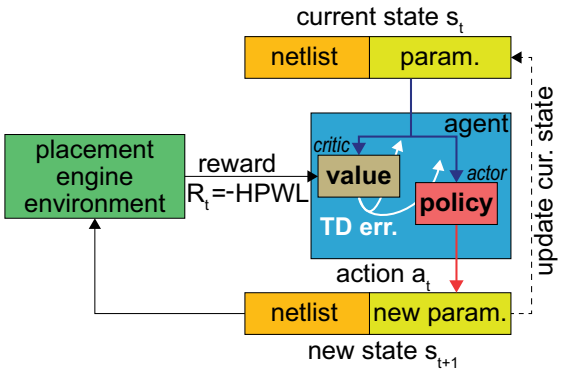

Figure 4: Actor-critic framework. The critic learns about and critiques the policy currently being followed by the actor.

a parameter setting may not be trivial or change significantly based on observation.

- Policy optimization often shows good convergence properties.

3.3 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 Figure 4. 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.

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 the Bellman equation:

$$v_\pi(s) = \mathbb{E}[R_{t+1} + \gamma v_\pi(s_{t+1}) | s_t = s] \tag{8}$$

which is used in temporal difference (TD):

$$\Delta \mathbf{w}_t = (R_{t+1} + \gamma v_{\mathbf{w}}(s_{t+1}) - v_{\mathbf{w}}(s_t)) \nabla_{\mathbf{w}} v_{\mathbf{w}}(s_t). \tag{9}$$

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:

$$\Delta \boldsymbol{\theta}_t = G_t \nabla_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(a_t|s_t). \tag{10}$$

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 $v_{\mathbf{w}}$:

$$G_t^{(n)} = \sum_{k=0}^{n-1} \gamma^k R_{t+k+1} + \gamma^n v_{\mathbf{w}}(s_{t+n}). \tag{11}$$

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:

$$\Delta \boldsymbol{\theta}_t = (G_t^{(n)} - v_{\mathbf{w}}(s_t)) \nabla_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(a_t|s_t). \tag{12}$$

The critic updates parameters $\mathbf{w}$ of $v_{\mathbf{w}}$ by n-step TD (Eq. 9) and the actor updates parameters $\boldsymbol{\theta}$ of $\pi_{\boldsymbol{\theta}}$ in direction suggested by critic by policy gradient (Eq. 12). In this work we use the *advantage actor-critic* method, called A2C [12], which was shown to produce

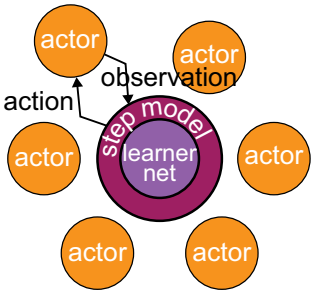

Figure 5: Synchronous parallel learner. The global network sends actions to the actors through the step model. Each actor gathers experiences from their own environment.

excellent results on diverse environments. As shown in Equation 12, an *advantage* function formed as the difference between returns and baseline state-action estimate is used instead of raw returns. The *advantage* can be thought of as a measure of how good a given action is compared to some average.

3.4 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) as well as the sparsity of data (there is no database of millions of netlists, placed designs or layouts). To solve both issues, we implement a parallel version of A2C, as depicted in Figure 5. In this implementation, an agent learns from experiences of multiple *Actors* interacting in parallel with their own copy of the environment. This configuration increases the throughput of acting and learning and helps decorrelate samples during training for data efficiency [6].

In parallel training setups, the learning updates may be applied synchronously or asynchronously. 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 weights of the network. One advantage is that it provides larger batch sizes, which is more effectively used by computing resources.

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

3.5 A 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 *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

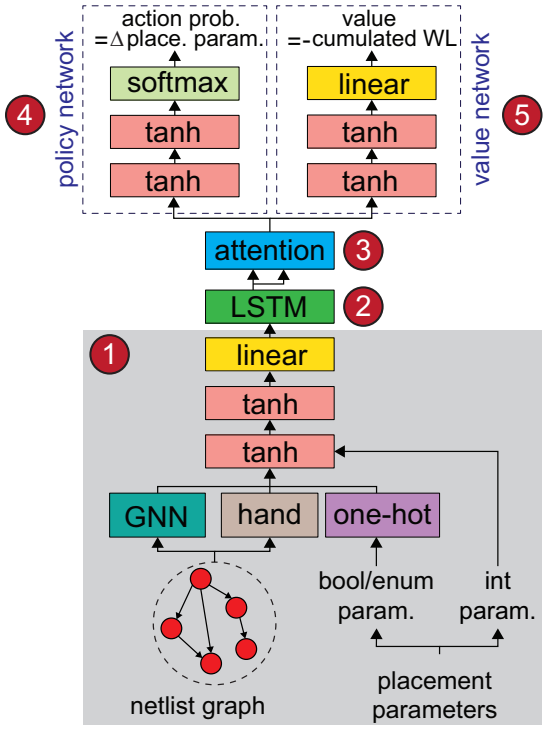

Figure 6: Overall network architecture of our agent. The combination of an LSTM with an Attention mechanism enables the learning of a complex recurrent optimization process. Table 4 provides the details of the sub-networks used here.

as follows as in [16]:

$$\begin{aligned} &\underbrace{(G_t^{(n)} - v_{\mathbf{w}}(s_t))\nabla_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(a_t|s_t)}_{\text{policy gradient}} + \\ &\underbrace{\beta\,(G_t^{(n)} - v_{\mathbf{w}}(s_t))\nabla_{\mathbf{w}} v_{\mathbf{w}}(s_t)}_{\text{value estimation gradient}} + \\ &\underbrace{\eta \sum_a \pi_{\boldsymbol{\theta}}(a|s_t) \log \pi_{\boldsymbol{\theta}}(a|s_t)}_{\text{entropy regularisation}}. \end{aligned} \tag{13}$$

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

The complete architecture of our deep neural network is shown in Figure 6. To compute value and policy, the concatenation of placement parameters with graph extracted features are first passed through two feed-forward fully-connected (FC) layers with *tanh* activations, followed by a FC linear layer. This is followed by a Long Short-Term Memory (LSTM) module with layer normalization and with 16 hidden standard units with forget gate. The feed-forward FC layers have no memory. Introducing an LSTM in the network, which is a recurrent layer, the model can base its actions on previous states. 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 [9] inspired from state-of-the-art

Table 4: Neural network parameters used in our RL agent architecture in Figure 6. The number of inputs of the first FC layer is as follows: 32 from GNN, 20 from Table 2, 24 one-hot encoding for the enum/bool types from Table 1, and 3 integer types from Table 1.

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)

Natural Language Processing architectures, to help the Recurrent Layer (RNN) focus on important 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:

$$\boldsymbol{a}_t(s) = \frac{\exp\left(\text{score}(\boldsymbol{h}_t, \bar{\boldsymbol{h}}_s)\right)}{\sum_{s'} \exp\left(\text{score}(\boldsymbol{h}_t, \bar{\boldsymbol{h}}_{s'})\right)} \tag{14}$$

where the alignment score function is:

$$\text{score}(\boldsymbol{h}_t, \bar{\boldsymbol{h}}_s) = \boldsymbol{h}_t^\top \boldsymbol{W_a} \bar{\boldsymbol{h}}_s. \tag{15}$$

The global context vector:

$$\boldsymbol{c}_t = \sum_s \boldsymbol{a}_t(s) \bar{\boldsymbol{h}}_s \tag{16}$$

is combined with the hidden state to produce an attentional hidden state as follows:

$$\tilde{\boldsymbol{h}}_t = \tanh\left(\boldsymbol{W_c}[\boldsymbol{c}_t \circ \boldsymbol{h}_t]\right). \tag{17}$$

This hidden state is then fed to the two heads of the network, both composed of two FC layers with an output softmax layer for the policy and an output linear layer for the value. The parameters of our network are summarized in Table 4.

3.6 Our Self-Play Strategy

Inspired from AlphaZero [13], our model learns without any supervised samples. We do not use expert knowledge to pre-train the network using good 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.

4 EXPERIMENTAL RESULTS

To train and test our agent, we select 15 benchmarks designs from OpenCores, ISPD 2012 contest and two RISC-V single cores, presented in Table 5. We use the first eleven for training and last four for testing. We synthesize the RTL netlists using *Synopsys Design Compiler*. We use *TSMC* 28nm technology node. The placements are done with *Cadence Innovus* 17.1. 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 pre-placed by hand. For successful placements, a lower bound of total cell area divided by floorplan area is set on parameter *max density*. IO pins are placed automatically by the tool between metals 4 and 6.

Table 5: Benchmark statistics based on a commercial 28nm 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
test set (unseen netlist)							
LDPC	39K	41K	4.1K	18	19	328.0	2
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

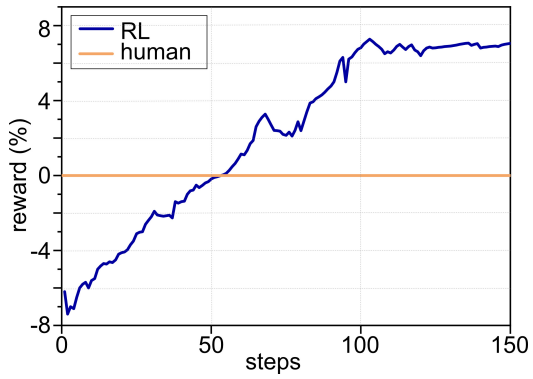

Figure 7: Training our agent for 150 iterations (= 14,400 placements). The reward is an aggregate reward from all training netlists. Training time is within 100 hours. Human baseline: reward = 0.

4.1 RL Network Training Setting

We define our environment using OpenAI Gym interface [2] and implement our RL agent network in Tensorflow. We use 16 parallel environments (16 threads) in our synchronous A2C framework.

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 Figure 7. We observe that the mean reward accross all netlists converges asymptotically to a value of 6.8%, meaning wirelength is reduced in average by 6.8%.

Training over 150 iterations (= 14,400 placements) takes about 100 hours. Note that 99% of that time is spent to perform the placements while updating the neural network weights takes less than 20min. Without parallelization, training over the same number of placements would take 16 × 100 hr = 27 days.

Table 6: Comparison of half-perimeter bounding box (HPWL) after placement on training netlists among human design, Multi-Armed Bandit (MAB) [1], and our RL-based method. HPWL is reported in m. Δ denotes percentage negative improvement over human design.

Netlist	human	MAB [1] (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%)

An explained variance of 0.67 shows 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 non-centered RMSProp as 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, totalling a mini-batch size of 96. All experiments use gradient clipping by norm to avoid exploding gradients (phenomenom common 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.

For a given environment, we select a random netlist and we always start with the corresponding human parameter set to form an initial state for each episode. Each environment is reset for episodes of 16 steps. Training on K environments in parallel, each performing a placement on a different netlist, the reward signal is averaged on the K netlist for the network updates, which decreases the reward variance and ultimately help the network generalize to unseen netlists as prescribed in [11].

4.2 Netlist Training Results

For comparison, we use the state-of-the-art tool auto-tuner OpenTuner [1] that we adapt for Innovus as baseline. In this framework, a Multi-Armed Bandit (MAB) selects at each iteration a search technique among Genetic Algorithm, Differential Evolution, Simulated Annealing, Torczon hillclimber, Nelder-Mead and Particle Swarm Optimization, based on a score that forces a trade-off between exploitation (use arm that worked best) and exploration (use a more rarely used arm). We run the search techniques in parallel, evaluating 16 candidate parameter sets at the same time. We run the tuner on the eleven training netlists and record the best achieved wirelength, performing 1,300 placements per netlist so that the total number of placements equals those of the RL agent training.

Table 6 shows the best wirelengths found by the MAB as well as RL agent during training. The human baseline is set by an experienced engineer who tuned the parameters for a day. We see that the RL agent outperforms MAB on most netlists, reducing HPWL by 9.8% on Rocket Core benchmark. All in all, both methods improve quite well on the human baseline.

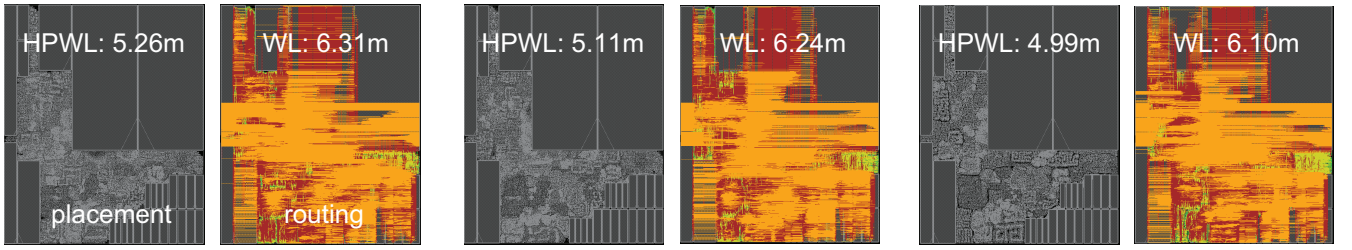

Figure 8: 28nm full-chip GDSII layouts of OpenPiton.

Table 7: Comparison on test netlists of best wirelength found (one iteration = one placement performed). HPWL is reported in m.

Netlist	human	#iter.	MAB [1]	#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 8: Best placement parameters found for Netcard benchmark.

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

4.3 Unseen Netlist Testing Results

To verify the ability of our agent to generalize we check its performance on the four unseen test netlists. Without any additional training (the network parameters are fixed), the RL agent iteratively improves a random initial parameter set by selecting action a with highest predicted $\pi_\theta(a)$ value, as described in Equation 5. Because our actions are deterministic, the resulting set of parameters is known, and fed back to the network. We repeat this process until the estimated value decreases for 3 consecutive updates and backtrack to the settings with highest value. This way a "good" candidate parameter set is found without performing any placement. We then perform a unique placement with that parameter set and record the obtained wirelength.

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 wirelength found, allotting 50 sequential iterations to the MAB.

The best wirelength found by our RL agent and the MAB on all four test netlists is show in Table 7. We observe our RL agent achieves superior wirelengths consistently, performing only one placement. Table 8 shows the best parameter settings found by MAB and RL agent on Netcard. Interestingly enough, we can see how the two optimizers found separate ways to minimize HPWL: WL driven vs. congestion driven.

Table 9: PPA comparison after routing on test set. The target frequencies are 1GHz, 500MHz, 833MHz, 666MHz from top to bottom.

ckt	metric	human	MAB [1]	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

4.4 PPA Comparison After Routing

To confirm improvement in HPWL after placement is translated into one in final routed wirelength, we perform routing of placed designs. They are all routed with same settings where metal layers 1 to 6 are used. The layouts of OpenPiton Core are shown in Figure 8. We verify target frequency is achieved and routing succeeded without congestion issues or DRC violations. PPA of routed designs is summarized in Table 9. We observe that HPWL reduction after placement is conserved after routing on all test designs, reaching 7.3% and 11% wirelength savings on LDPC and Netcard compared with the human baseline. Footprints are 74,283 um^2 for LDPC, 1,199,934 um^2 for OpenPiton, 728,871 um^2 for Netcard and 894,115 um^2 for Leon3.

5 CONCLUSIONS

Our placement parameter optimizer based on deep RL provides preset of improved parameter settings without human intervention. We believe this is an important step to shift from the "CAD" mindset to the "Design Automation" mindset. We use a novel representation to formulate states and actions applied to placement optimization. Our experimental results show our agent generalizes well to unseen netlists and consistently reduces wirelength compared with a state-of-the-art tool auto-tuner, in only one iteration without any additional training.

REFERENCES

[1] J. Ansel et al. OpenTuner: An Extensible Framework for Program Autotuning. PACT '14.
[2] G. Brockman et al. OpenAI Gym, 2016.
[3] C. Bron and J. Kerbosch. Algorithm 457: Finding All Cliques of an Undirected Graph. *Commun. ACM*, 1973.
[4] V. Colizza et al. Detecting rich-club ordering in complex networks. *Nature Physics*, 2006.
[5] O. Coudert. Exact Coloring Of Real-life Graphs Is Easy. DAC '97.
[6] L. Espeholt et al. IMPALA: Scalable Distributed Deep-RL with Importance Weighted Actor-Learner Architectures, 2018.
[7] W. Hamilton et al. Inductive Representation Learning on Large Graphs. NIPS '17.
[8] R. B. Lehoucq and D. C. Sorensen. Deflation Techniques for an Implicitly Restarted Arnoldi Iteration. *SIAM J. Matrix Analysis Applications*, 17:789–821, 1996.
[9] M.-T. Luong et al. Effective Approaches to Attention-based Neural Machine Translation.
[10] I. L. Markov et al. Progress and Challenges in VLSI Placement Research. ICCAD '12.
[11] A. Mirhoseini et al. Chip Placement with Deep Reinforcement Learning, 2020.
[12] V. Mnih et al. Asynchronous Methods for Deep Reinforcement Learning, 2016.
[13] D. Silver et al. Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm, 2017.
[14] R. E. Tarjan. Depth-First Search and Linear Graph Algorithms. *SIAM J. Comput.*, 1:146–160, 1972.
[15] L. van der Maaten and G. E. Hinton. Visualizing Data using t-SNE. 2008.
[16] O. Vinyals et al. StarCraft II: A New Challenge for Reinforcement Learning, 2017.
[17] D. J. Watts and S. H. Strogatz. Collective dynamics of 'small-world' networks. *Nature*, 1998.
[18] C. Xu et al. A Parallel Bandit-Based Approach for Autotuning FPGA Compilation. FPGA '17.
[19] C. H. Yu et al. S2FA: An Accelerator Automation Framework for Heterogeneous Computing in Datacenters. DAC '18.