Reinforcement Learning Driven Physical Synthesis
(Invited Paper)

Zhuolun He, Lu Zhang, Peiyu Liao, Yuzhe Ma, Bei Yu
Department of Computer Science and Engineering
The Chinese University of Hong Kong
{zlhe,lzhang,pyliao,yzma,byu}@cse.cuhk.edu.hk

***Abstract*—Physical synthesis has emerged as a core component in a modern circuit design flow. Large-scale optimization problem is often involved in the process, which requires substantial efforts to solve and no optimality is guaranteed. Reinforcement learning provides one direction to deal with the above issue by automatically acquiring knowledge through experience, which has shown great success in various applications. In this paper, we introduce the foundation of and the progress in reinforcement learning, and review some recent approaches in applying reinforcement learning to physical synthesis. We hope to inspire more work and to see more talented ideas in this field.**

I. Introduction

Physical synthesis has undoubtedly emerged as a critical component in modern electronic design automation (EDA) flow. The primary purpose of physical synthesis is to perform timing closure [1]: given a netlist generated by logical synthesis, the goal of physical synthesis is to create a placed layout that satisfies the timing constraints, while other objectives like performance, power, and area (PPA) got optimized. The above process often involves solving large-scale sophisticated optimization problems, most of which are considered unlikely to have algorithms to solve in polynomial time (i.e. NP-hard). As the technology node scaling down, the design complexity is even increased. Besides the growth in problem size, the timing model also becomes more complicated, since the wire delay continually gets significant, which gradually dominates the overall performance. Given that, the demand is highlighted to develop accurate and efficient physical synthesis tools.

Recently, we see a clear trend of incorporating machine learning techniques into the field of EDA. In principle, machine learning is well suited to deal with signals for which no explicit mathematical formulation emerges due to unknown data distribution [2]. Therefore, one way to apply machine learning to the EDA flow is to substitute computationally expensive subprocesses with machine learning based (fast) approximations, where no new algorithm is derived. For example, machine learning models are utilized for hotspot detection [3], routability prediction [4], and sub-resolution assist feature (SRAF) insertion [5], respectively. Basically, these methods fall into the category of supervised learning, where the algorithm is designed to learn from labeled training data. In other words, expert knowledge is assumed to be known. However, the expert knowledge may not be sufficient since lots of state-of-the-art algorithms are hand-crafted heuristics, which have little or no performance guarantee. In this sense, we certainly hope to explore the algorithmic design space and obtain better performing behaviours.

Reinforcement learning, one of the basic paradigms in machine learning, has achieved great success in a wide range of disciplines including robot control [6], game playing [7], and natural language processing [8]. The power of reinforcement learning partly comes from its generality: by training an agent to optimize its actions through interactions with the environment, reinforcement learning can in principle create new knowledge about the space that the agent interacts with. Compared with supervised learning, no labeled data (input-output pairs) is required in reinforcement learning, nor the explicit correction of the sub-optimal actions. Instead, the target of an agent is to maximize numerical rewards given by the environment. We will formally discuss the learning paradigm in subsequent sections.

The remainder of the paper is organized as follows: Section II recapitulates the preliminaries of reinforcement learning, Section III introduces recent attempts of applying reinforcement learning to physical synthesis, and Section IV concludes the paper with discussions.

II. Preliminaries

A. Basis of Reinforcement Learning

Typically, Reinforcement Learning is formalized as a *Markov Decision Process* (MDP), a 4-tuple (S, A, P, R), where

- S is a set of states to describe the environment;
- A is a set of actions that the agent can take;
- P is the transition probability between states under actions;
- R is the immediate reward signal.

MDP is a discrete-time stochastic control process: at timestamp t, the agent is at a state s_t; then the agent selects an action $a_t \in A$, and finds itself in a new state s_{t+1}, with an immediate reward r_{t+1}. By concatenating states, actions, and rewards of all time steps together, we obtain a sequence called an episode: $s_0, a_0, r_1, s_1, a_1, \cdots$, which fully describes the trajectory of the agent.

The objective in MDP is to find a good policy $\pi(s)$ to maximize the accumulative reward (return)

$$G_t = \sum_t^{\infty} \gamma^t r_t, \tag{1}$$

where $\gamma \in [0, 1]$ is the discount factor to penalize future rewards. We define the value of a state s under policy π (denoted as $V_\pi(s)$) as the expected return starting from that state. According to *Bellman Equation*, the state value can be stated recursively:

$$\begin{aligned} V_\pi(s) &= \mathbb{E}[G_t | S_t = s] \\ &= R(s, \pi(s)) + \gamma \sum_{s'} P(s, \pi(s), s') V_\pi(s'). \end{aligned} \tag{2}$$

Similarly, we define the expected return from taking an action a in state s under policy π as the action value:

$$\begin{aligned} Q_\pi(s, a) &= \mathbb{E}[G_t | S_t = s, a_t = a] \\ &= R(s, a) + \gamma \sum_{s'} P(s, a, s') V_\pi(s') \\ &= R(s, a) + \gamma \sum_{s'} P(s, a, s') \mathbb{E}_{a' \sim \pi} Q_\pi(s', a'). \end{aligned} \tag{3}$$

B. Common Learning Approaches

1) Dynamic Programming: If the model is fully known (i.e., P and R of the MDP is known), and both the state space and action space are finite, solutions to the MDP can be obtained through iterative policy improvement following bellman equation. The *generalized policy iteration* (GPI) algorithm refers to the optimization process consisting of iterative policy evaluation (through Equation (2)) and policy improvement by

greedily selecting the action with highest action value:

$$\pi'(s) = \underset{a}{\operatorname{argmax}} \left\{ R(s,a) + \gamma \sum_{s'} P(s,a,s') V_\pi(s') \right\}. \quad (4)$$

Through alternating policy evaluation and policy improvement,

$$\pi_0 \xrightarrow{\text{evaluate}} V_{\pi_0} \xrightarrow{\text{improve}} \pi_1 \xrightarrow{\text{evaluate}} \cdots \xrightarrow{\text{improve}} \pi^* \xrightarrow{\text{evaluate}} V^*,$$

the policy is guaranteed to be better:

$$\begin{aligned} & Q_\pi(s, \pi'(s)) \\ =& Q_\pi(s, \underset{a}{\operatorname{argmax}} \left\{ R(s,a) + \gamma \sum_{s'} P(s,a,s') V_\pi(s') \right\}) \\ =& \max_a Q_\pi(s,a) \geq Q_\pi(s, \pi(s)). \end{aligned} \quad (5)$$

2) Monte Carlo Methods: Monte Carlo methods rely on large amount of repeated random sampling to obtain statistical properties, which are broadly used in various optimization and simulation problems. Following the GPI framework, policy evaluation can be substituted by Monte Carlo simulations, while policy improvement remains the greedy behaviour.

Recall from Equation (2) that $V_\pi(s) = \mathbb{E}[G_t | S_t = s]$, we can empirically estimate G_t through sampling instead of exactly computing the expectation:

$$v_\pi(s) = \frac{\sum_{t=1}^T \mathbb{1}[s_t = s] G_t}{\sum_{t=1}^T \mathbb{1}[s_t = s]}, \quad (6)$$

where $\mathbb{1}[\cdot]$ is the indicator function. Similarly we can estimate $Q_\pi(s,a)$ as well:

$$q_\pi(s,a) = \frac{\sum_{t=1}^T \mathbb{1}[s_t = s, a_t = 1] G_t}{\sum_{t=1}^T \mathbb{1}[s_t = s, a_t = a]}. \quad (7)$$

With sufficiently amount of samples, the procedure is able to precisely estimate the values due to the law of large numbers.

In Monte Carlo methods, state values and action values are estimated through raw experience without knowing the model dynamics, which is often the case in reality. However, it is important to note that Monte Carlo methods only work in episodic problems (finite episode length), otherwise the return may not be correctly computed. In practice, the method is often regarded as sample inefficient, in that it requires complete episodes to update estimates.

3) Temporal Difference Learning: Similar to Monte Carlo methods, temporal difference (TD) Learning is another model-free learning algorithm that learns from raw experiences. The key difference between the two methods is that TD is based on bootstrapping: the values are estimated with regard to other estimates, rather than exclusively relying on actual rewards.

From bellman equation, we know that $R(s_t, \pi(s_t)) + V_\pi(s_{t+1})$ is an unbiased estimation of state value $V_\pi(s_t)$. This motivates the following update rule:

$$v_\pi(s_t) = v_\pi(s_t) + \alpha(R(s_t, \pi(s_t)) + \gamma v_\pi(s_{t+1}) - v_\pi(s_t)), \quad (8)$$

where α is the step size (a.k.a. learning rate), and $R(s_t, \pi(s_t)) + \gamma v_\pi(s_{t+1})$ is refereed to as the TD target. Similarly, we can estimate $Q_\pi(s_t, a_t)$ as well:

$$\begin{aligned} q_\pi(s_t, a_t) &= q_\pi(s_t, a_t) \\ &+ \alpha(R(s_t, a_t) + \gamma q_\pi(s_{t+1}, a_{t+1}) - q_\pi(s_t, a_t)). \end{aligned} \quad (9)$$

The algorithm that uses Equation (9) to estimate action value in GPI is known as SARSA in the literature.

A popular algorithm, Q-learning [9], is an off-policy variant of TD learning. In an off-policy algorithm, the policy that the agent follows to interact with the environment (behavior policy) is independent of the optimal policy that the agent aims to learn (target policy). To see the difference, in SARSA (or Equation (9)), a_{t+1} is selected following the target policy π, which the algorithm is going to estimate. In Q-learning, however, the action is greedily selected, despite the fact that the target policy π is actually not greedy:

$$\begin{aligned} q_\pi(s_t, a_t) &= q_\pi(s_t, a_t) \\ &+ \alpha(R(s_t, a_t) + \gamma \max_a q_\pi(s_{t+1}, a) - q_\pi(s_t, a_t)). \end{aligned} \quad (10)$$

It is interesting to note that taking max over all actions may overestimate the action values. Please refer to [10] for more discussions.

TABLE I Comparisons of value-based reinforcement learning algorithms. Methods differ in whether it samples and whether it boostraps.

Method	Sampling?	Bootstrapping?
Dynamic Programming	No	Yes
Monte Carlo Method	Yes	No
Temporal Difference Learning	Yes	Yes
Brute-force	No	No

So far, we have introduced three value-based reinforcement learning approaches. As shown in TABLE I, we compare the three algorithms, together with brute-force, in terms of whether it samples and whether it boostraps.

4) Policy Gradient: Instead of estimating state or action values and deciding policy according to the values, an alternative approach is to directly search in the policy space. Suppose the policy is parameterized by θ, the objective is still to maximize the accumulative reward:

$$J(\theta) = \mathbb{E}[V_\theta(s_0)]. \quad (11)$$

As by Policy Gradient Theorem [11], we have:

$$\nabla_\theta J(\theta) = \mathbb{E}_{\pi_\theta}[Q_\pi(s,a) \nabla \ln \pi(a|s;\theta)]. \quad (12)$$

The algorithm REINFORCE [12] can be regarded as the Monte Carlo policy gradient. Recalling that $Q_\pi(s_t, a_t) = \mathbb{E}[G_t | s_t, a_t]$, REINFORCE estimates $\nabla_\theta J(\theta)$ by $G_t \nabla \ln \pi(a_t|s_t;\theta)$ and updates the policy parameters by

$$\theta = \theta + \alpha \gamma^t G_t \nabla \ln \pi(a_t|s_t;\theta), \quad (13)$$

for each $t = 1, \cdots, T$ in the episode.

5) Actor Critic: In vanilla policy gradient algorithms, the variance of the gradient could be large. To stabilize the training, one idea is to utilize the value model to guide the policy search, namely the actor-critic [13] framework. In short, the critic is trained to predict the state or action values using TD learning, and the actor is trained to select actions by policy gradient using the predicted values.

C. State-of-the-art Algorithms

A great many variants of the above basic reinforcement learning approaches have been proposed in recent years. Common techniques include experience replay [14] and prioritized experience replay [15] to both break sample correlations and to improve sample efficiency, multi-step learning [16] to accelerate training, trust region optimization [17] to constrain step direction, and maximum entropy reinforcement learning [18] to encourage exploration.

There is no silver bullet in reinforcement learning algorithm selection. However, the first thing to consider is often in the action space. For discrete action space, one may consider deep Q-learning (DQN) [19] and its variants(e.g. rainbow [20] that combines lots of useful techniques), or advantage actor-critic (A2C) [21] and its variants (e.g. actor-critic with experience replay (ACER) [22]). For continuous action space, one popular choice is soft actor-critic (SAC) [23], and other alternatives include proximal policy optimization (PPO) [24] and Twin-delayed DDPG (TD3) [25]. Among the above, A2C and PPO are on-policy algorithms while the rests are off-policy. In summary, lots of the newly proposed algorithms aim to reduce variance to stabilize training.

III. Reinforcement Learning Driven Physical Synthesis

A. Placement

1) Macro Placement: Reinforcement Learning has been used in chip macro placement [26]. In this work, the target is to place a netlist graph of macros (e.g. SRAMs) to a chip canvas to optimize PPA. Overall, the method is to first sort the macros in descending size, which are placed sequentially by the agent, and followed by force-directed standard cell clusters placement and greedy legalization. Therefore, in their reinforcement learning settings, each state is a possible partial placement, and each action is to place a macro onto the canvas. The reward is always 0 except the last action (that leads to the terminal state), where the reward is the negative weighted sum of proxy wirelength and congestion of the placement.

Distilling useful features from state representation is essential for good decision making. In the policy network, the netlist is encoded with a graph convolutional network (GCN) for macro embedding and edge embedding, which are concatenated with other metadata. A key intuition presented in the paper is that a policy network capable of transferring placement knowledge across chips should first be able to encode the features into a meaningful signal in inference time. Given that, they proposed to first train the model for reward prediction in a supervised manner, with a dataset of 10000 chip placements of 5 netlists. After the supervised training, the reward prediction layers are removed, and the encoder component is used in the policy network.

To handle different grid size, they allow a maximum row/column number of 128, and the L-shaped unused section are masked if the grid size is smaller. The decoder is composed of deconvolution layers and batch normalization layers to generate a probability distribution for the actions. The RL agent is trained with PPO [24] with a clipped objective:

$$L^{\text{CLIP}}(\theta) = \hat{\mathbb{E}}_t[\min(r_t(\theta)\hat{A}_t, \text{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t)], \quad (14)$$

where $\hat{\mathbb{E}}_t$ is the expected value at time step t, r_t is the probabilistic ratio between new policy and old policy, and $\hat{A}_t$ is the estimated advantage at time step t.

During the evaluation, the policy network without further tuning yields a placement result (which they called zero-shot placement) in less than one second. They observed that a pre-trained model with 2 hours fine-tuning performs even better than another model trained 24 hours from scratch, which indicates the effectiveness of offline training. In comparison with baseline methods, the RL method converges in 3 to 6 hours, which is much faster than simulated annealing (18 hours) and slower than RePlAce [27] (1 to 3.5 hours), one of the state-of-the-art placer. They also compared with human experts (a chip design team) that involves many iterations over a few weeks. When it comes to the placement quality, the RL method is able to consistently meet timing and congestion constrains, and outperforms human experts in several metrics including WNS, area, power, and wirelength.

2) Placement Heuristic Selection: Another work worth mentioning on RL-enhanced placement is [28]. In this work, the RL agent is trained to select the move type in an SA framework for Field-Programmable Gate Arrays (FPGA) placement. This is naturally modeled as a multi-arm bandit problem with a Q table for each action (possible move type). Given that, the agent selects the move type with the highest Q-value and samples a random move, which the SA engine decides to accept or reject. The proposed technique was integrated into VTR 8 placer [29] and achieved improved quality-runtime tradeoff.

B. Routing

1) Global Routing: Global routing partitions the routing region into tiles and decides tile-to-tile paths for all nets while attempting to optimize some given objective function (e.g., total wirelength and circuit timing) [30]. In [31], RL is used for sequential single net routing. In their settings, the agent observes a vector representing current coordinate, target pin coordinate, and the available resource of the neighbouring tracks, and picks a direction (out of the 6 possible) to move. The reward is $+100$ if the target pin is reached, otherwise -1 is given.

Since the action space is discrete and finite, they selected to use DQN [19] for training. Before the exploration by RL agent, they collect samples by running A* algorithm to fill the experience replay buffer (which they called memory burn-in). The evaluation was conducted on toy examples like $8 \times 8 \times 2$ grid graphs with only a few nets. One interesting observation is that A* performs much better in easy cases (i.e., no edge with positive capacity is fully utilized in A*), while RL agent wins more in difficult cases.

2) Detailed Routing: Given global routing paths, detailed routing determines the exact tracks and vias for nets [30]. In a track-assignment detailed router, a track assignment step is performed before detailed routing to place the long routes onto tracks defined by the width space pattering (WSP), which reduces the search space for the router as it only needs to connect the components (i.e., instance terminal) of the same net together. In [32], RL is used to decide the track assignment order.

Specifically, the attention-based encoder-decoder framework [33] is adopted, where greedy rollout is used as a baseline to train the agent with REINFORCE [12]. The observation of the agent includes an overlap graph of instance terminals, in which an edge indicates two terminals overlap in x-range and hence cannot be assigned to the same track, and an assignment graph, which is a bipartite graph of instance terminals and available tracks.

The evaluation was done on two artificial datasets of analog design problems. The small dataset contains placement solutions for Comparators and OpAmp, which consists of 10-100 instance terminals, while the large dataset are designs for analog-to-digital converter (ADC) with 100–1000 instance terminals. In comparison with the genetic algorithm (GA), the RL solution runs $100\times$ to $1000\times$ faster, while GA outperforms in solution quality. The paper also shows that the performance of RL agent could be further improved with more training samples.

3) Printed Circuit Boards Routing: To target printed circuit boards (PCBs) routing, [34] proposed to use Monte-Carlo-tree-search (MCTS) to guide the agent training. Basically, a search tree is constructed based on rollouts to obtain optimal solution, which is then back-propogated to update the agent parameters. The input of the agent is the grid graph (a matrix) states, and the output is one of the four actions representing the four directions to go.

In practice, they consider a single layer board of 30×30, so the model architecture is a convolutional neural network (CNN). To train this model, a dataset of 2000 randomly generated circuits (grid size 30-by-30) routed with maze routing and manual routing was created. Then each vertex on the net of the circuit can be taken as the head of a training sample. With one sample taken from each net, totally 9459 samples were obtained. Experiments then show that the agent is able to obtain good solutions after thousands of training iterations, while some instances could be successfully routed by A* or Lee's algorithm (because of the routing order).

The authors have also emphasized to use maximal reward in MCTS expansion (Max-UCT), instead of using average reward (Avg-UCT) as in vanilla MCTS. Experimental results show that using Max-UCT indeed greatly outperforms Avg-UCT in terms of wire redundancy ratio and convergence speed. One possible explanation, as pointed out by the authors, is that the routing search space is very large and nice solutions are rarely seen, which makes average reward somehow useless in the context.

4) Sizing: Transistor sizing can be modeled as a continuous parameter

search problem. In [35], a sequence to sequence model is used to encode observations (e.g. voltage at a node) and to decide sizing solution (e.g. width and length of transistors, capacitance of capacitors). Reward function was designed to include both hard constraints and optimization targets. To train the model, DDPG [36] was used, with truncated uniform distribution as noise for search space exploration. Experiments on two transimpedance amplifiers circuits show that the proposed method can achieve comparable performance with much higher (250×) sample efficiency compared with grid search based human design. It also outperforms Bayesian Optimization under the same runtime constraint.

Later on, GCN was involved [37] since the circuit topology can be naturally represented by a graph, whose vertices are transistors and edges are wires. In this work, the agent determines the action for each node (e.g., width, length, and multiplexer for an NMOS), and therefore the action decoder is component-specific. The state for each node includes one-hot representation for transistor index and component type, as well as a vector of selected features. Note that the action space is continuous here, otherwise the action space will be too large to deal with. The reward is still figure of merits (FoM), a weighted sum of normalized performance metrics. The effectiveness of RL agent is demonstrated on a few real-world circuits, compared with black-box optimization tools and human experts. Experiments on knowledge transfer between nodes and between topologies are also conducted, which shows that transfer learning indeed helps, and using GCN for feature extraction is critical.

IV. Conclusion

Significant progress has been made in solving challenging problems across various domains using Reinforcement Learning, and physical synthesis also benefits from it. In this paper we reviewed a few of these attempts targeting some core problems (placement, routing, sizing) in physical synthesis. We see that impressive results are obtained thanks to state-of-the-art training algorithms and model architectures. To further make use of the great power of reinforcement learning, we argue that new problem formulation is critical, in order to align the large-scale optimzation problem in physical synthesis to the optimal control problem in reinforcement learning.

References

[1] C. J. Alpert, S. K. Karandikar, Z. Li, G.-J. Nam, S. T. Quay, H. Ren, C. N. Sze, P. G. Villarrubia, and M. C. Yildiz, "Techniques for fast physical synthesis," *Proceedings of the IEEE*, vol. 95, no. 3, pp. 573–599, 2007.

[2] Y. Bengio, A. Lodi, and A. Prouvost, "Machine learning for combinatorial optimization: a methodological tour d'horizon," *arXiv preprint arXiv:1811.06128*, 2018.

[3] D. Ding, B. Yu, J. Ghosh, and D. Z. Pan, "EPIC: Efficient prediction of IC manufacturing hotspots with a unified meta-classification formulation," in *Proc. ASPDAC*, 2012, pp. 263–270.

[4] C.-W. Pui, G. Chen, Y. Ma, E. F. Young, and B. Yu, "Clock-aware ultrascale fpga placement with machine learning routability prediction," in *Proc. ICCAD*, 2017, pp. 929–936.

[5] H. Geng, W. Zhong, H. Yang, Y. Ma, J. Mitra, and B. Yu, "SRAF insertion via supervised dictionary learning," *IEEE TCAD*, 2019.

[6] S. Gu, E. Holly, T. Lillicrap, and S. Levine, "Deep reinforcement learning for robotic manipulation with asynchronous off-policy updates," in *Proc. ICRA*, 2017, pp. 3389–3396.

[7] D. Silver, J. Schrittwieser, K. Simonyan, I. Antonoglou, A. Huang, A. Guez, T. Hubert, L. Baker, M. Lai, A. Bolton *et al.*, "Mastering the game of go without human knowledge," *Nature*, vol. 550, no. 7676, pp. 354–359, 2017.

[8] D. He, Y. Xia, T. Qin, L. Wang, N. Yu, T.-Y. Liu, and W.-Y. Ma, "Dual learning for machine translation," in *Proc. NIPS*, 2016, pp. 820–828.

[9] C. J. Watkins and P. Dayan, "Q-learning," *Machine learning*, vol. 8, no. 3-4, pp. 279–292, 1992.

[10] R. S. Sutton and A. G. Barto, *Reinforcement learning: An introduction*. MIT press, 2018.

[11] R. S. Sutton, D. A. McAllester, S. P. Singh, and Y. Mansour, "Policy gradient methods for reinforcement learning with function approximation," in *Proc. NIPS*, 2000, pp. 1057–1063.

[12] R. J. Williams, "Simple statistical gradient-following algorithms for connectionist reinforcement learning," *Machine learning*, vol. 8, no. 3-4, pp. 229–256, 1992.

[13] V. R. Konda and J. N. Tsitsiklis, "Actor-critic algorithms," in *Proc. NIPS*, 2000, pp. 1008–1014.

[14] L.-J. Lin, "Self-improving reactive agents based on reinforcement learning, planning and teaching," *Machine learning*, vol. 8, no. 3-4, pp. 293–321, 1992.

[15] T. Schaul, J. Quan, I. Antonoglou, and D. Silver, "Prioritized experience replay," *arXiv preprint arXiv:1511.05952*, 2015.

[16] R. S. Sutton, "Learning to predict by the methods of temporal differences," *Machine learning*, vol. 3, no. 1, pp. 9–44, 1988.

[17] J. Schulman, S. Levine, P. Abbeel, M. Jordan, and P. Moritz, "Trust region policy optimization," in *Proc. ICML*, 2015, pp. 1889–1897.

[18] B. D. Ziebart, "Modeling purposeful adaptive behavior with the principle of maximum causal entropy," 2010.

[19] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller, "Playing atari with deep reinforcement learning," *arXiv preprint arXiv:1312.5602*, 2013.

[20] M. Hessel, J. Modayil, H. Van Hasselt, T. Schaul, G. Ostrovski, W. Dabney, D. Horgan, B. Piot, M. Azar, and D. Silver, "Rainbow: Combining improvements in deep reinforcement learning," in *Proc. AAAI*, 2018.

[21] V. Mnih, A. P. Badia, M. Mirza, A. Graves, T. Lillicrap, T. Harley, D. Silver, and K. Kavukcuoglu, "Asynchronous methods for deep reinforcement learning," in *Proc. ICML*, 2016, pp. 1928–1937.

[22] Z. Wang, V. Bapst, N. Heess, V. Mnih, R. Munos, K. Kavukcuoglu, and N. de Freitas, "Sample efficient actor-critic with experience replay," *arXiv preprint arXiv:1611.01224*, 2016.

[23] T. Haarnoja, A. Zhou, P. Abbeel, and S. Levine, "Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor," *arXiv preprint arXiv:1801.01290*, 2018.

[24] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, "Proximal policy optimization algorithms," *arXiv preprint arXiv:1707.06347*, 2017.

[25] S. Fujimoto, H. Van Hoof, and D. Meger, "Addressing function approximation error in actor-critic methods," *arXiv preprint arXiv:1802.09477*, 2018.

[26] A. Mirhoseini, A. Goldie, M. Yazgan, J. Jiang, E. Songhori, S. Wang, Y.-J. Lee, E. Johnson, O. Pathak, S. Bae *et al.*, "Chip placement with deep reinforcement learning," *arXiv preprint arXiv:2004.10746*, 2020.

[27] C.-K. Cheng, A. B. Kahng, I. Kang, and L. Wang, "Replace: Advancing solution quality and routability validation in global placement," *IEEE TCAD*, vol. 38, no. 9, pp. 1717–1730, 2018.

[28] K. E. Murray and V. Betz, "Adaptive fpga placement optimization via reinforcement learning," in *Proc. MLCAD*, 2019, pp. 1–6.

[29] V. Betz and J. Rose, "Vpr: A new packing, placement and routing tool for fpga research," in *Proc. FPL*. Springer, 1997, pp. 213–222.

[30] L.-T. Wang, Y.-W. Chang, and K.-T. T. Cheng, *Electronic design automation: synthesis, verification, and test*. Morgan Kaufmann, 2009.

[31] H. Liao, W. Zhang, X. Dong, B. Poczos, K. Shimada, and L. Burak Kara, "A deep reinforcement learning approach for global routing," *Journal of Mechanical Design*, vol. 142, no. 6, 2020.

[32] H. Liao, Q. Dong, X. Dong, W. Zhang, W. Zhang, W. Qi, E. Fallon, and L. B. Kara, "Attention routing: track-assignment detailed routing using attention-based reinforcement learning," *arXiv preprint arXiv:2004.09473*, 2020.

[33] W. Kool, H. Van Hoof, and M. Welling, "Attention, learn to solve routing problems!" *arXiv preprint arXiv:1803.08475*, 2018.

[34] Y. He and F. S. Bao, "Circuit routing using monte carlo tree search and deep neural networks," *arXiv preprint arXiv:2006.13607*, 2020.

[35] H. Wang, J. Yang, H.-S. Lee, and S. Han, "Learning to design circuits," *arXiv preprint arXiv:1812.02734*, 2018.

[36] D. Silver, G. Lever, N. Heess, T. Degris, D. Wierstra, and M. Riedmiller, "Deterministic policy gradient algorithms," 2014.

[37] H. Wang, K. Wang, J. Yang, L. Shen, N. Sun, H.-S. Lee, and S. Han, "Gcn-rl circuit designer: Transferable transistor sizing with graph neural networks and reinforcement learning," *arXiv preprint arXiv:2005.00406*, 2020.