Guiding FPGA Detailed Placement via Reinforcement Learning

by

Pooria Esmaeili Taheri

A Thesis

presented to

The University of Guelph

In partial fulfilment of requirements

for the degree of

Master of Science

in

Engineering

Guelph, Ontario, Canada

ABSTRACT

Guiding FPGA Detailed Placement via Reinforcement Learning

Pooria Esmaeili Taheri
University of Guelph, 2022

Advisor:
G. Grewal and S. Areibi

Field Programmable Gate Array (FPGA) placement is an important optimization step within the CAD implementation flow. Several metrics have to be optimized during placement including wirelength, congestion, timing, and most important routability. This thesis proposes to use Reinforcement Learning to further enhance the solution quality provided by the detailed placement stage. We show that a well-trained RL agent can capture different characteristics of each circuit placement and use these features in the decision-making process. The proposed RL framework targets simultaneously two objectives in the form of wirelength and external pin count to further improve circuit routability. Results obtained indicate the proposed RL-enhanced algorithm can reduce the runtime by 50% while achieving comparable solution quality to the GPlace3.0 detailed placement flow. These results can be further improved by refinement of the RL-enhanced algorithm which motivates further exploration of integrating RL techniques in the FPGA CAD flow.

Acknowledgements

I would like to take this opportunity to express my sincere appreciation and thanks to my advisor Dr. Shawki Areibi and Dr. Gary Grewal for their constant support and guidance throughout my research and writing this thesis. I also would like to thank my colleagues in the ISLAB: Timothy Martin, Mohsen Fathi, and Roni Mittal. Their help and collaboration paved the way to make this work possible. Last but not least, I want to thank my family for being present in every stage of my life. Thank you for being so incredible.

To

my family

whose love and encouragement helped accomplish this

thesis.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

A *Field Programming Gate Array* (FPGA) is an integrated circuit that can be configured to execute a specific computing task. FPGAs introduced in the 80's filled the gap between high-performance *Application-Specific Integrated Circuit* (ASICs) and flexible General Purpose Processors (GPPs) since designers were capable of programming and reprogramming them and also mapping algorithms into hardware via Hardware Description Languages. FPGAs are growing in size and architectural complexity as indicated by Moore's law, putting pressure on the runtime of FPGA Computer-Aided Design (CAD) algorithms [3]. FPGAs became widely adopted due to their re-programmability for fast prototyping and in-system debugging of a hardware design. However, this advantage is undermined when the CAD flow runtime is excessive or the Quality of Results (QoR) is poor. Therefore, developing CAD tools that are capable of providing fast and near-optimal solutions is necessary for the FPGA industry [4].

The FPGA CAD flow consists of several steps including synthesis, placement, and routing of the design on target FPGA. Each of these steps requires solving a large-scale

complex optimization problem, known to be NP-hard in which the optimal solution cannot be obtained in polynomial time. Therefore, modern CAD tools rely heavily on heuristic approaches to solve these problems with acceptable QoR in a reasonable time. However, designing such heuristic algorithms requires a huge amount of time as well as extensive knowledge about the CAD flow and FPGA architecture. Besides, an important step in designing these algorithms is parameter tuning which greatly depends on the designer's intuition and experience and required extensive experimentation. This tuning process is so crucial that it can lead to 70% variation in the QoR without changing the source code or target architecture [5].

Within the CAD flow, placement is a crucial stage since it is usually the most time-consuming stage of the FPGA compilation flow [6], (taking several hours or even days for the large designs). Besides, it highly affects the QoR of the entire flow. A poor placement solution can result in long routing cycles or even unroutability since FPGA routing resources are pre-fabricated and constrained. Moreover, the maximum frequency in which the final design can operate is also affected by the final placement solution provided by the tool.

The placement problem is defined as assigning logic blocks of a circuit netlist to legal locations on the FPGA while optimizing one or more objectives. Placements are evaluated with different metrics such as wirelength (WL), external pin (EP) count, timing, congestion, and more important routability. The placement procedure consists of two main stages: Global placement, and detailed placement. During global placement, cells are spread across the FPGA in a way that an objective like wirelength or circuit delay is optimized. Then, during detailed placement, local refinements are made to further optimize the design objectives like timing and routability.

There are three main approaches to tackle the placement problem: partitioning-based (PB), simulated annealing (SA), and analytical placement (AP). Partition-based algorithms [7] employ divide-and-conquer techniques to break down the problem into smaller placement problems. The runtime of these techniques is short, but they suffer from poor QoR. Simulated annealing placers [8] were widely used due to their robustness and suitability for handling multiple objectives. However, these methods are often very non-scalable and as the size and complexity of designs are increasing, their runtime is getting prohibitively long. Analytical-based algorithms, like GPlace3.0 [1] and elfplace[9], obtain the desired locations of logic blocks using mathematical techniques in a way that optimize an objective like wirelength, or routability under a set of constraints that are gradually refined. While these methods have high scalability, they typically need a detailed improvement stage to fully optimize an FPGA design. These detailed refinements are often performed using either low temperature simulated annealing or greedy block moves, as in GPlace3.0 [10] and RippleFPGA [11]. However, these methods do not take different characteristics of each circuit into the decision-making process and use the same static flow for different designs.

As mentioned earlier, FPGA CAD flow relies on heuristic algorithms to solve complex optimization problems which necessitate substantial effort and time to manually tune different parameters. Data-driven algorithms, such as Machine Learning (ML), can be employed to automate this process. ML is a subset of Artificial Intelligence (AI) that aims to learn the structure and patterns in the data without any hard-coded program or human intervention where it can generalize what it has learned to a new problem. These qualities have encouraged EDA researchers to employ ML to overcome the above-mentioned limitations and challenges. There have been several attempts

that adopt ML-based frameworks to create smarter EDA tools and achieve better performance, [5, 12, 13].

More recently, a branch of ML known as Reinforcement Learning (RL) has received more attention. In Reinforcement Learning frameworks, an agent learns the optimal behavior through experience to optimize an objective by taking different actions in interacting with its environment and receiving feedback about its performance via a reward signal. The RL agent's goal is to maximize its cumulative reward. RL can be used to solve a wide range of problems that are involved with making a sequence of decisions, such as robotics [14] and chess [15]. Because the placement problem can be formulated as a sequence of cell moves to explore a large solution space, it is a good candidate to benefit from RL.

Several stages of the FPGA CAD flow have suffered from traditional static implementations. These implementations do not take the different characteristics of the circuits into account and go through the same flow for all circuits, resulting in sub-optimal solutions or long design cycles. In addition, the circuit is constantly changing during the flow and a dynamic implementation can help to reach an optimal solution. Recent works presented at [4] and [16], integrated RL in a Simulated Annealing based FPGA placement algorithm to enable it to dynamically adapt to the specific problem being solved as the optimization progresses. Work in [17] targets another stage of FPGA CAD flow. It uses the RL technique to speed up the routing process in FPGA design flow. However, none of these works has attempted to integrate RL in the detailed placement stage of the FPGA CAD flow.

In this thesis, we propose to integrate an RL framework into the detailed placement phase of state-of-the-art GPlace3.0 [1]. We then develop several RL models to cap-

ture the different characteristics of each circuit placement and use them in the decision-making process. Our goal here is to attain a smarter placement flow by considering the attributes of each placement solution to guide the placer to produce a better sequence of moves as it goes through the placement instead of following a static flow.

1.1 Motivation

There are several motivations and advantages for integrating reinforcement learning into the static GPlace3.0 detailed placement flow:

1. In the current static flow, the optimization technique is performed across the whole placement with the same objective and it overlooks the different characteristics of each part of the placement. Moreover, the order of objectives and number of iterations the optimization technique is applied for each objective are statically defined a priori before the placement process.

2. This same static flow is applied to the different circuits which may not be the best flow for each one of them as they have different attributes. Therefore, the ability to dynamically identify the best optimization strategy during the placement can improve the QoR.

3. As the size and complexity of FPGA grows, placement runtime is exceedingly increasing. Using an intelligent algorithm rather than a static flow with a fixed number of iterations can guide the placer tool to achieve the same QoR in less runtime.

1.2 Contributions

The main contributions of this thesis are:

1. We integrate an RL framework into the GPlace3.0 detailed placement phase to dynamically choose the best optimization strategy during the detailed placement. We propose to divide the placement into a grid of regions in order to capture the different characteristics of each part of the placement.

2. We develope several RL models including Tabular Q-Learning, Deep Q-Learning, and Advantage Actor-Critic to select the most effective move type as optimization progresses.

3. The proposed RL framework is able to reduce the CPU time by 50% while obtaining QoR comparable to the state-of-the-art GPlace3.0 across the 2016 ISPD benchmark suite [18].

4. The proposed RL framework is applied to a multi-objective optimization problem with conflicting objectives. The framework is able to achieve comparable QoR (without sacrificing either of the objectives) and yet cut the CPU time.

1.3 Thesis Organization

The reminder of this thesis is organized as follows: Necessary and relevant background that is necessary to understand the thesis is introduced in Chapter 2. This includes the description of the FPGA placement problem and objectives, the target FPGA architecture, GPlace3.0 flow, the benchmark used, and necessary background on Reinforcement

Learning. In chapter 3, we present the previous approaches for using reinforcement learning in CAD flow. We first present the previous works on integrating RL in the FPGA CAD flow. Next, we introduce the works that target ASIC CAD flow. The proposed methodology is described in Chapter 4. We first present the GPlace3.0 detailed placement flow. Then describe how we integrate RL in the detailed placement. Finally, we introduce several RL models investigated in this thesis. In chapter 5, we present the result of each RL model and compare them to the GPlace3.0 detailed placement. Chapter 6 summarizes the results obtained and gives different directions that can be persued in the future.

Chapter 2

Background

In this Chapter, the necessary background for understanding this thesis is introduced. Section 2.1 explains the FPGA placement problem and its constraints in detail alongside the performance metrics of placement including wirelength and external pins. In section 2.2, we describe the FPGA device architecture we are targeting in this thesis. GPlace3.0 and its flow are presented in section 2.3. In section 2.4, we present the benchmarks that were used to evaluate our proposed RL frameworks. Finally, Reinforcement Learning concepts and techniques are introduced in section 2.5.

2.1 FPGA Placement

Within the FPGA CAD flow, placement is one of the most important and time-consuming step. Having a good placement is crucial as it affects the QoR of the entire flow. A bad placement solution can put lots of pressure on the router. Moreover, the clock frequency of the final design is highly dependent on the quality of placement. FPGA Placement

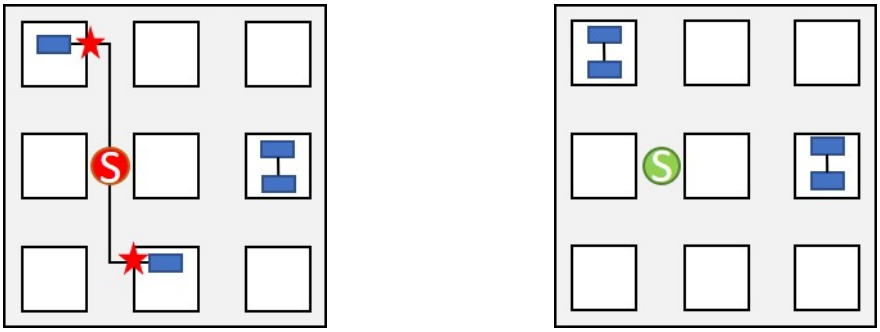
problem is defined as assigning logic blocks of a netlist to legal locations on the FPGA while optimizing one or more objectives like estimated wirelength, timing, or routability. Wirelength is usually the most important objective for a placement problem as smaller wirelength can lead to less circuit delay and fewer wire resources consumption which has a great impact on placement routability. This importance motivated the exploration of new methods for FPGA placement that lead to using analytical-based approaches. Analytical-based placement is an important class of placement algorithms that obtain the desired locations of logic blocks using mathematical techniques in a way that minimizes an objective like wirelength, or circuit delay while it ignores non-overlapping constraints. Analytical methods are widely used in ASIC placement because the solutions produced by analytical placers are non-integers which are acceptable in ASIC placement as cells can be placed anywhere across the die. However, using analytical placers is more challenging in FPGA placement due to FPGAs pre-fabricated and discrete architecture that imposes the placement solutions to be integers. Analytical placement procedure consists of three stages: Global placement, legalization, and detailed placement. During global placement, cells are spread on the chip in a way that an objective like wirelength or circuit delay is optimized. During this step, overlaps are permitted, therefore an additional legalization step is necessary. The objective of legalization is to produce a legal and non-overlapping solution while minimizing the total displacement between the original placement and legal placement. During detailed placement, some other optimization techniques are used to further optimize placement objectives.

2.1.1 Half-Perimeter Wirelength (HPWL)

There are many different metrics to evaluate FPGA placements such as wirelength (WL), congestion, and routability. Among these metrics, WL is usually the main objective of the placers. However, there are different models to calculate WL. HPWL is one of the models that is widely used. The HPWL of a circuit is calculated by measuring the HPWL of all the nets of the placement and then summing them up. The HPWL model of a net is defined by the width and height of the smallest rectangle enclosing all nodes in the net [19]. To calculate the HPWL, we just need to add the width and the length of the rectangle together.

2.1.2 External Pins

The FPGA architecture has a limited amount of global resources for routing the wires in the circuit. It is therefore beneficial to limit the congestion of these resources by grouping together LUTs and FFs that have many connections to each other. This helps to ensure that the placement can be successfully routed. External Pins (EP) is a metric that can be used to measure this degree of clustering. When LUTs and FFs are grouped into the same CLB their connections can be made internally. External pins measure how many connections need to be made to resources *outside* of each CLB, and therefore how much demand is placed on the global routing resources. Figure 2.1 is an example of external pins optimization. Figure 2.1a is showing two connected logic blocks that are placed in different CLBs and they are utilizing a switch block to make the connection. Figure 2.1b shows the logic blocks after external pin optimization. Both of the logic blocks are placed within the same CLB and therefore, there is no need to utilize a switch block anymore.



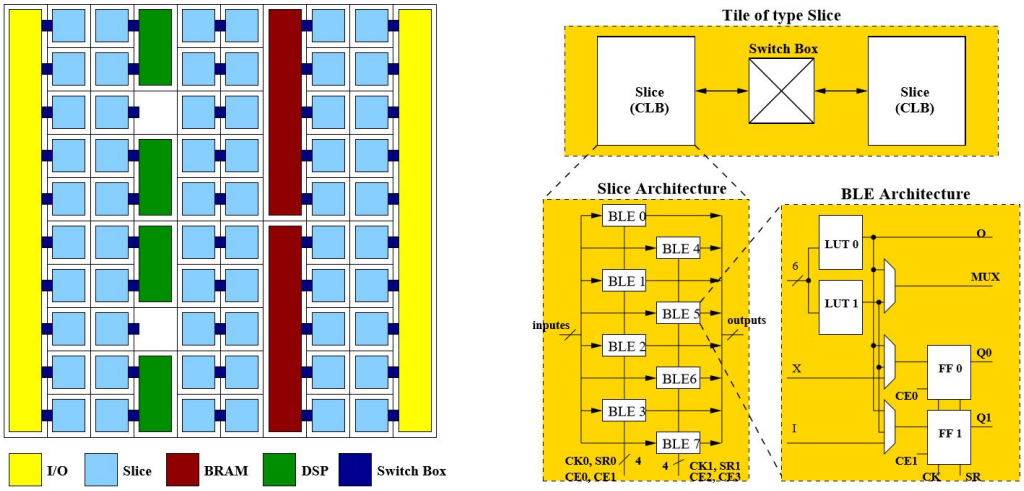
(a) External Pins Before Optimizing. (b) External Pins After Optimizing.

Figure 2.1: External Pins Optimization Example.

2.2 FPGA Architecture

In this thesis, we target the modern Xilinx Ultrascale VU095 FPGA device architecture [20] shown in Figure 2.2a. This is a heterogeneous architecture which means there are different types of logic blocks such as Slices, Random Access Memory Blocks (BRAMs), Digital Signal Processing blocks (DSPs), and I/Os. Each Slice in this architecture contains one Configurable Logic Block (CLB). Architecture of a CLB is illustrated in Figure 2.2b. Each CLB can have up to 8 Basic Logic Elements (BLEs). BLEs consist of 1-2 Lookup Tables (LUTs) and 2 Flip Flops (FFs) to enable the implementation of both combinational and sequential circuits. A LUT is a memory that maps input values to output values and is used to implement simple operations. FF is a register that is used to store data. Logic blocks are surrounded by prefabricated routing segments in both vertical and horizontal directions. In addition to these routing segments, there are programmable switches that can be configured to connect different logic blocks across the

FPGA chip by forming a path of routing segments.



(a) UltraScale Layout.

(b) Tile Architecture of Slice Type.

Figure 2.2: Xilinx UltraScale Architecture [1].

2.3 GPlace3.0 Flow

GPlace3.0 [1] is an analytical-based placer targeting Xilinx UltraScale FPGAs that tries to optimize both wirelength and routability. GPlace3.0 flow has three main phases as is illustrated in Figure 2.3. In phase I, GPlace3.0 tries to produce a high-quality placement with minimizing wirelength. It should be noted that GPlace3.0 avoids early packing in the placement process since initial packing can prevent later optimizations and flexibilities. Instead, it first performs a pin propagation pre-placement (Figure 2.3: step (1)) to place logic blocks near to their I/Os. Then, several iterations of WL-driven global placement (Figure 2.3: step (2)) are performed. Each iteration consists of an analytic

flat placement and window-based legalization. During analytic placement, GPlace3.0 employs the Star+ [21] wirelength model to obtain coordinates of each block while it tries to optimize the wirelength without considering non-overlapping constraints. Then, during window-based legalization, overlaps are reduced by spreading logic blocks.

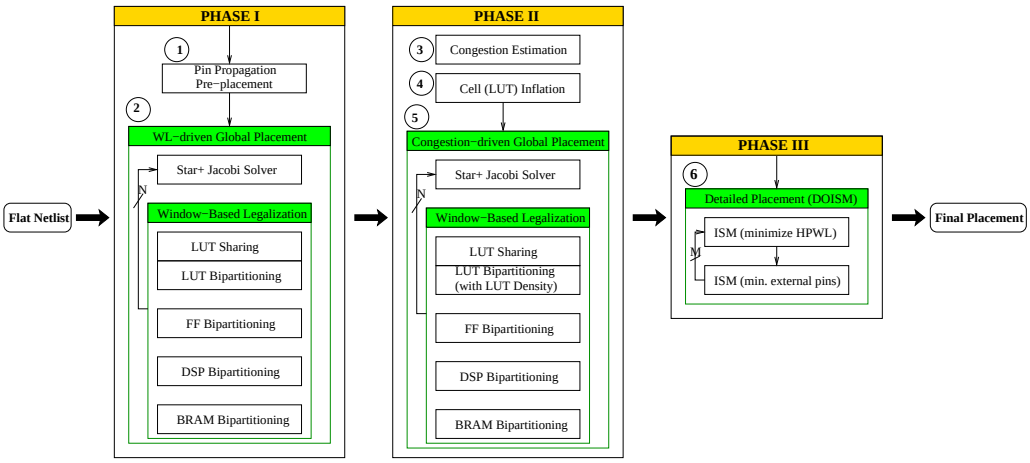


Figure 2.3: GPlace3.0 flow [1]

Although the placement produced at end of phase I is optimized for wirelength and satisfies all hard constraints, there is no guarantee that the placement is free of congestion or even routable. Therefore, in Phase II, an efficient global routing is performed to identify congested regions (Figure 2.3: step (3)) and then LUTs are inflated (Figure 2.3: step (4)). Followed by these steps, a congestion-driven placement is performed which shares the same framework with the WL-driven global placement except that LUTs are inflated and bi-partitioned based on their densities during legalization.

In phase III, GPlace3.0 employs a detailed placement algorithm (Figure 2.3: step (6)), called DOISM, to further optimize the placement in terms of wirelength and external pin count. DOISM is based on Independent-Set Matching (ISM) [22] that uses bipartite

matching to improve the wirelength. GPlace3.0 employs the same idea to reduce the external pin count to improve the routability. We describe this phase in more detail in section 4.1

2.4 Benchmarks

To test our proposed frameworks, we used the 12 benchmarks from ISPD 2016 placement contest [18], shown in Table 2.1. The range of key features for these 12 benchmarks is shown in Table 2.1. These features vary in a wide range, providing us with a rich set of benchmarks. All of the figures presented in the Result chapter of this thesis belong to FPGA-10 to keep things uniform across different models. The reason for choosing this benchmark is that this benchmark has a rich set of logic blocks as well as control signals.

Benchmark	#LUTs	#FF	#BRAM	#DSP	#CSet[a]	#IO	R.E[b]
FPGA-1	49K	55K	0	0	12	150	0.4
FPGA-2	98K	74K	100	100	121	150	0.4
FPGA-3	245K	170K	600	500	1281	400	0.6
FPGA-4	245K	172K	600	500	1281	400	0.7
FPGA-5	246K	174K	600	500	1281	400	0.8
FPGA-6	345K	352K	1000	600	2541	600	0.6
FPGA-7	344K	357K	1000	600	2541	600	0.7
FPGA-8	485K	216K	600	500	1281	400	0.7
FPGA-9	486K	366K	1000	600	2541	600	0.7
FPGA-10	346K	600K	1000	600	2541	600	0.6
FPGA-11	467K	363K	1000	400	2091	600	0.7
FPGA-12	488K	602K	600	500	1281	400	0.6

Table 2.1: ISPD 2016 Placement Contest Benchmark Statistics

[a]#CSet: Combination of reset and control-enable signals in the benchmark

[b]Rent Exponent (RE)

#LUTs	#FF	#BRAM	#DSP	#CSet	#IO	R.E
44K-518K	52K-603K	0-1035	0-620	11-2684	150-600	0.4-0.8

Table 2.2: Range of Key Circuit Features

2.5 Reinforcement Learning

Reinforcement Learning (RL) is a branch of machine learning that can be applied to a wide variety of problems that are associated with sequence decision making such as robotics [14], Chess [15], and real-time strategy games [23]. RL problems are formalized as a Markov Decision Process (MDP), consisting of four key elements:

- **State space (S)**: a set of possible states of the environment and agent;
- **Action Space (A)**: a set of actions that agent can take;
- **Transition Probabilities (P)**: the transition probability between states under actions;
- **Reward (r)**: the immediate reward for taking an action in a state;

In RL problems, an agent learns through experience by interacting with an environment to maximize its cumulative reward. An overview of the RL problem is depicted in Figure 2.4. At time step t, the agent interacts with the environment by taking an action a_t from the set of possible actions A. Then, at the next time step $t + 1$, the agent interprets the reaction of the environment and determines the reward of its action r_{t+1} and the next state s_{t+1} from a set of available states S. Based on the previous experiences, the agent selects a new action a_{t+1} to perform to maximize its cumulative reward over time. An RL agent may include one or more of these components:

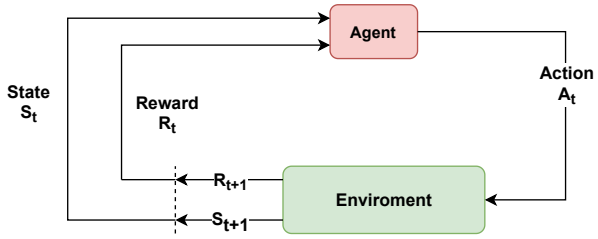


Figure 2.4: Reinforcement Learning Problem

- **Model**: agent's representation of the environment;
- **Policy** π: agent's behavior function;
- **State Value function V(s)**: how good is each state;
- **Action Value function Q(s, a)**: how good is each state-action pair;

RL can be mainly divided into Model-based and Model-free methods. Model-based RL methods, such as dynamic programming, assume that the transition probabilities of the states are known. The agent then can calculate the optimal actions using the model directly. However, model-based methods are not practical as the state and action space grow. Model-free RL, such as Temporal Difference (TD) learning, learns directly from experience and does not care about the inner dynamics of the environment. So it does not need to store all the states and actions combinations.

Model-free RL methods can be divided into two categories: 1. Value-Based RL, 2. Policy-Based RL. Value-based methods learn the state or state-action values. With having these values, the agent can derive the optimal behavior in every state, known as optimal policy, to maximize its cumulative reward. The policy-based methods learn

directly the policy function that maps state to action instead of learning a value function and choosing the action based on it.

We first introduce the main components and concepts of reinforcement learning through section 2.5.1 to 2.5.5. Then, we introduce a well-known value-based method in section 2.5.6 as well as a policy-based method in section 2.5.7. In section 2.5.8, we introduce the advantages and disadvantages of both value-based and policy-based methods. Finally, in section 2.5.9, we discuss the potential challenges in incorporating RL in optimization problems.

2.5.1 Discounted Expected Return

As we said before, the objective of the RL agent is to find a policy that maximizes its cumulative expected (discounted) reward, known as return R_t. The return R_t is defined as the sum of the rewards from the current state s_t up to the terminal state at time T. Equation 2.1 shows how the return is calculated where γ is the discount factor and r_t is the immediate reward at time step t. The discount factor determines the importance of the immediate reward versus the future reward. Higher values of γ emphasize the future rewards, whereas the lower values care more about the immediate reward.

$$R_t = \sum_{\tau=t}^{T} \gamma^{\tau-t} r_\tau \tag{2.1}$$

2.5.2 Policy and Value Functions

The policy function π determines how the agent should behave given a state. It produces a probability distribution $\pi(a|s)$ over the actions in state s and then the agent samples the

next action from that probability distribution.

The value function $V(s)$ defines how good it is to be in state s. The $V_\pi(s)$ estimates the expected discounted return starting from state s, and then following policy $\pi(s)$, equation 2.2.

$$V_\pi(s) = \mathbb{E}[R_t|s_t = s] \tag{2.2}$$

The action-value function $Q(s, a)$ defines how good it is to take action a in state s. The $Q_\pi(s, a)$ estimates the expected discounted return starting from state s, taking action a, and following policy π, equation 2.3.

$$Q_\pi(s, a) = \mathbb{E}[R_t|s_t = s, a_t = a] \tag{2.3}$$

2.5.3 Temporal Difference (TD) Learning

Temporal difference learning is a model-free reinforcement learning method that updates its policy at every time step t in contrast to Monte Carlo methods which update the policy at the end of the episode where the true return is known. In TD methods, the true return is unknown and is approximated with the TD-target, equation 2.4. Therefore, there is no need to wait until the end of the episode. The TD-target is the sum of the immediate reward and discounted expected value of the next state. In this technique, we use the estimated value of subsequent states to determine the value of the current state, which is known as bootstrapping.

$$\text{TD-target} = r_{t+1} + V(s_{t+1}) \tag{2.4}$$

Algorithm 1 [2], presents the TD-method. In each iteration of each episode, an action a_t is sampled according to policy π (line 6). The action is performed and the agent receives the immediate reward r_{t+1} and transitioned to new state s_{t+1} (line 7). Then, the value-table V is updated with the difference of the TD-target and $V(s_t)$, known as TD-error (line 8).

Algorithm 1 TD Learning

1: Algorithm parameters: step size $\alpha \in (0, 1]$
2: Initialize $V(s)$, for all $s \in \mathcal{S}$, arbitrarily except that $V(\text{terminal}) = 0$
3: **for** each episode **do**
4: Initialize S_t
5: **while** S is not terminated **do**
6: $A_t \leftarrow$ action given by π for S_t
7: Take action A_t, observe R_{t+1}, S_{t+1}
8: $V(S_t) \leftarrow V(S_t) + \alpha[r_{t+1} + \gamma V(S_{t+1}) - V(S_t)]$
9: $S_t \leftarrow S_{t+1}$
10: **end while**
11: **end for**

2.5.4 Policy Gradient Methods

Policy Gradient methods directly parameterize the policy $\pi(a|s, \theta)$ instead of learning the value function. The quality of each policy can be evaluated by the policy's score function $J(\theta)$, equation 2.5. The objective is to learn a policy that maximizes the scalar value of $J(\theta)$, equation 2.6.

$$J(\theta) = \mathbb{E}\left[\sum_{\tau} R(\tau)\pi_\theta(\tau)\right] \tag{2.5}$$

$$\theta^*(s) = \arg\max_{\theta} J(\theta) \tag{2.6}$$

The policy's parameters θ are updated using the gradient ascent. The gradient ascent is similar to the gradient descent but, it takes the direction of the steepest increase in the score function, equation 2.7. The derivative of $J(\theta)$ can be determined by applying the Policy Gradient Theorem that has been derived in [2], equation 2.8.

$$\theta_{t+1} = \theta_{t+1} + \alpha \nabla_\theta J(\theta_t) \tag{2.7}$$

$$\nabla J(\theta) = \mathbb{E}\Big[\sum_\tau R(\tau) \nabla_\theta log \pi_\theta(a_t|s_t)\Big] \tag{2.8}$$

2.5.5 Exploration vs Exploitation

Model-free RL algorithms need to establish a balance between exploration and exploitation. Exploration is more of a long-term benefit concept where it allows the agent to improve its knowledge about each action which could lead to long-term benefit. Exploitation basically exploits the agent's current estimated value and chooses the greedy approach to get the most reward. A stochastic policy allows the agent to explore the state space without always taking the same action. As a consequence, it handles the exploration/exploitation trade-off without hard coding it. However, deterministic policies require an exploration mechanism. One such method is ϵ-greedy. In this method, the agent takes the greedy action (exploit) with probability $1 - \epsilon$, and takes an action at random (explore) with probability ϵ. ϵ can be either a fixed parameter or can be adjusted according to a schedule.

2.5.6 Q-Learning

Q-learning is a model-free, value-based RL algorithm that seeks to learn the optimal Q-values for each state-action pair by iteratively improving the estimated Q-function. The Q-learning algorithm is built upon the TD learning method and used the TD-target concept to improve its estimates. The action-value update rule of Q-learning is illustrated in equation 2.9. The Q-learning uses the same algorithm presented at 1, but the update rule (line 8) is replaced by the equation 2.9.

Q-learning is known as an off-policy algorithm (Sutton & Barto, 1998) since the target can be computed using the experience collected by following a different policy.

$$Q(s_t, a_t) = Q(s_t, a_t) + \alpha[r_t + \gamma \max_{a_{t+1}} Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)] \tag{2.9}$$

2.5.7 Actor-Critic

Actor-Critic models are policy gradient methods that make use of the value function to learn the policy parameters θ. The actor-critic algorithms have a hybrid architecture that combines policy-based and value-based RL methods into a single model. Figure 2.5 shows the architecture of the Actor-Critic Models. There are 2 interacting models in this architecture. An Actor (policy-based) decides which action should be taken given a state as well as a Critic (value-based) who evaluates how good was the selected action. The Critic estimates the value function. This could be the action-value function, Q(s, a), or state-value function, V(s). The Actor updates the policy distribution in the direction suggested by the Critic. As we train our model, the actor learns to take better actions from critic feedback, and the critic learns to provide better feedback.

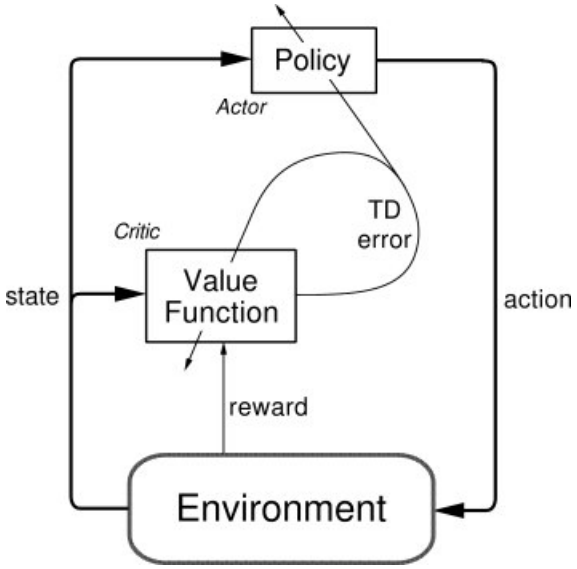


Figure 2.5: Actor-Critic Architecture [2]

Actor-Critic models use the TD-error to update both networks. The critic learns the Q-values by parameterizing the Q-function $Q_w(s, a)$ and estimates the expected value of the current state and the next state that contributes to the TD error, equation 2.10. Then the critic's output drives all the learning in both actor and critic. The actor uses the critic's output as an estimate of the total return $R(\tau)$ and as a result, the $R(\tau)$ in equation 2.8 is replaced by $Q_w(s, a)$. The critic and the actor update rules are presented in equations 2.11, and 2.12, respectively. In the actor-critic models, learning is on-policy meaning that the critic learns the value function for one policy while following it.

$$\text{TD-error}(\delta_t) = r_{t+1} + Q_w(s_{t+1}, a_{t+1}) - Q_w(s_t, a_t) \tag{2.10}$$

$$w_{t+1} = w_{t+1} + \alpha\delta_t \nabla_w Q_w(s_t, a_t) \tag{2.11}$$

$$\theta_{t+1} = \theta_{t+1} + \alpha Q_w(s_t, a_t) \nabla_\theta log \pi_\theta(a_t|s_t) \tag{2.12}$$

2.5.8 Policy-Based vs Value-Based

One advantage of policy-based RL is that the policy is directly updated at each time step leading to smooth improvement of policy. However, In value-based RL, the value function is updated at each time step and thus a small change in the value function can make a significant change to the policy. As a result, the policy-based methods usually have a more stable convergence property.

Another advantage of the policy-based methods is their effectiveness in high-dimensional or continuous state-action space. Since the action can be estimated directly, there is no need to identify the Q-value for each possible discrete action.

Furthermore, the policy-based RL can learn stochastic policies whereas the value-based methods usually come up with a deterministic policy. A deterministic policy can lead to a sub-optimal solution, especially in partially observable environments.

The main disadvantage of policy-based RL is that they usually converge to a local rather than the global optimum [2]. Besides, the policy-based methods usually suffer from high variance returns.

2.5.9 RL Challenges

In real-world problems, the agent cannot have a complete perception of the environment, known as partial observability which makes the learning process difficult. Moreover, a realistic environment can be non-stationary which means that an action can be rewarding

in a particular state while it can be diminishing later on. It is hard for the agent to keep track of all these changes. Besides, to solve an optimization problem using RL, different methods can be employed and each of them is associated with a set of parameters. Choosing the right method and parameters configuration is usually very challenging and requires lots of trial and error.

One crucial part of each RL method is defining the reward function as it is the only feedback that the environment gives to the agent. Reward function becomes more important as the number of objectives arises, especially when these objectives have conflicts with each other. Another challenge that we already described is exploration vs. exploitation. To build an optimal policy, it is necessary to balance the exploration-exploitation trade-off which is different based on the problem.

2.6 Summary

In this chapter, we first introduced the FPGA placement problem and its challenges. Then, we described target FPGA architecture and its constraints. Besides, we presented the GPlace3.0 flow in detail. Finally, we introduced Reinforcement Learning concepts and models that are used in this thesis. In Chapter 3, several works that have tackled different steps of the FPGA and ASIC CAD flow are discussed.

Chapter 3

Literature Review

In this chapter, we first discuss the works that attempted to apply RL techniques in different steps of FPGA CAD flow in section 3.1. Later, in section 3.2, we cover the works that target improving ASIC CAD flow using RL-based methods.

3.1 RL in FPGA CAD Flow

Machine Learning and specifically supervised and unsupervised learning have been integrated within several FPGA CAD flows including [24], [25], and [26]. However, only a few papers have proposed the integration and usage of RL within the FPGA CAD flow. In this section, we describe the different papers that used RL in both the FPGA placement and Routing problems.

3.1.1 RL In FPGA Placement

The work in [16] targets the global placement stage of FPGA CAD flow. It is built upon the VTR [27] in which they use a Q-learning method combined with ϵ-greedy strategy to choose the best move among random moves for Logic block, RAM, DSP, and IO. The reward is given after each move as the amount of improvement in HPWL. In this work, they only have a single state as their state space. Their results show they could achieve the same quality of results as VTR in 50% less time.

The work in [4] is built upon [16] with more actions and states. The current work has two states, exploration and exploitation, and the actions space now has seven directed moves and random moves. This time they use a softmax function for action selection which enables exploitation of multiple move types. They could achieve a 5-11% reduction in wire length and a 33-50% reduction in runtime compared to VTR.

3.1.2 RL In FPGA Routing

Work [17] targets FPGA routing problem. Their results show that their RL-based technique can obtain similar QoR compared to the conventional negotiated congestion-driven routing technique [24] with 30% speed up. Their objective through routing is to minimize the number of resource conflicts and to guide the agent towards this objective, they define their reward signal as the change in the number of conflicts. To train the agent, they use an action-value estimation method. They maintain a table of action values for each node and update this table after making each move. However, they do not give any explanation about their state and action space.

3.2 RL in ASIC CAD Flow

Various works aimed to employ machine learning in the ASIC CAD flow. Work [28] trains a model to predict the number of Design Rule Check (DRC) violations for a given macro placement. Works [29], and [30] use supervised learning techniques to predict the detailed routing routability and the detailed routing congestion, respectively. Recently, some works proposed to apply RL within the ASIC CAD flow to improve the QoR and runtime. In this section, we present the works that are integrating and using RL techniques in ASIC placement and routing problems.

3.2.1 RL In ASIC Placement

The work in [31] focuses on the ASIC placement and, specifically, the global placement phase. In this work, all the possible placements during the flow are a state of the agent. The agent uses an Asynchronous Advantage Actor-Critic (A3C) algorithm to choose the best action. Actions space is defined as the direction of the moves in the placer and parameters in the objective function. The reward is given to the agent when the target density is reached as the amount of improvement in HPWL. They were able to improve HPWL by 1% compared to the traditional flow.

The work in [32] focuses on the ASIC chip placement. The RL agent sequentially maps the macros of a netlist while trying to optimize wirelength, congestion, power, timing, and area. They train a deep neural network using Proximal Policy Optimization (PPO) to estimate the actions based on the current state. Each possible partial placement of macros is defined as a state while the actions involve moving macros. After each action, the reward is given to the agent as the weighted sum of reduction in wirelength

and congestion. The RL agent could achieve 10X less runtime compared to the manual flow.

The work [33] is another work that targets ASIC placement. They propose to train an RL agent to optimize the placement parameters in order to minimize wirelength. In this work, each netlist and its current parameter set is considered to be a single state. Therefore, they first encode the netlist information using a mixture of graph handcrafted features and graph neural network embeddings. This information can help the agent to generalize the tuning process. Moreover, a change in a subset of parameters is considered an action, and the wirelength improvement is the reward associated with that action. Their result shows that a trained RL agent can improve wirelength up to 11% and 2.5% on unseen netlists.

3.2.2 RL In ASIC Routing

The work in [34] targets the global routing in the CAD flow. The problem is stated as a Markov decision process, and they use a deep Q-learning algorithm to train the agent. As they are training the network, state transitions are stored in a replay buffer to be used for updating the network weights through back-propagation. In this work, the state is defined as a 12-dimensional vector. This vector shows the location of the agent, distance to the target pin, and information of all edges that the agent can cross. The actions are the direction of the moves from the current place. The reward is a strongly positive value when finding the shortest path and is -1 when failed.

[35] tries to use RL in the detailed routing stage to find optimal net ordering. In this work, a state is represented by collective features of all nets, and an action is shown with a real number vector which each number in the vector is defining an ordering score

of a net. Reward signal is the subtraction of the total cost achieved by the traditional approach from the total cost achieved by the agent's action. They train multiple A3C agents in parallel so they can explore the environment with different policies and then, these agents update the global network asynchronously. They have shown using this approach, the number of DRC violations is reduced by 14% and 0.7% less total cost compared to Dr.CU 2.0 [36] detailed router

The work presented in [37], targets standard cell routing. They first generate an initial routing candidate and then using RL, design rules violations are fixed incrementally. In this work, the environment provides the agent about the violations and the agent learns how to fix them. To train their network, they use PPO which is a policy gradient-based RL algorithm. The state of the environment is represented by the multi-layer grid space of the sticks and action is choosing one of the grids to be routed. The environment provides the agent with two rewards. The first one is a negative reward associated with the number of steps that force the agent to finish the job as soon as possible. The second one is associated with the number of DRC improvements, encouraging the agent to reduce the number of DRC violations as much as possible. Using this approach, they were able to route a cell that was considered to be unroutable manually, reducing the cell size by 11%.

To Rip or not to Rip [38] is another work that focuses on global routing in integrated circuits. In this work, they first create an initial routing solution. Then, they use the rip-up and reroute (RRR) scheme to reduce the number of overflows in the whole design. Their idea is to use an RL agent to decide whether it should rip and reroute a net or not. They use an actor-critic based PPO method to train their agent since training an action-value function is tedious due to the high dimensionality of action space. States are represented with a feature vector of the net n including wirelength of the net, overflow degree, and

competition degree. Possible actions for each net are to rip-up and reroute it or skip it. The reward function is a linear equation of improvement in wirelength, via numbers, and short areas before and after the action. Using this method, they were able to reduce the number of rerouted nets effectively. However, they have not reported the runtime for different methods.

3.3 Summary

In this chapter, we presented previous attempts of improving FPGA and ASIC CAD flow using RL approaches. None of the above works target the detailed placement phase of FPGA placement. Moreover, they mainly focus is only on one objective while we try to benefit from RL in a multi-objective problem. In Chapter 4, we first present the detailed placement flow of GPlace3.0. Then we describe our proposed framework for using RL algorithms. Furthermore, we introduce different RL models investigated in this thesis.

Chapter 4

Methodology

In this chapter, the proposed methodology is introduced and clearly explained. First, the GPlace3.0 detailed placer ISM is presented in section 4.1. In section 4.2, the RL framework that integrates within the ISM detailed placer is introduced. Next, the reinforcement learning main components are presented in section 4.3. Finally, the RL models that were investigated in this thesis including Tabular Q-Learning , Deep Q-Learning, and Advantage Actor-Critic are explained in sections 4.4, 4.5, and 4.6 respectively. The result of each model and comparison will be introduced in the next chapter.

4.1 GPlace3.0 ISM

GPlace3.0 employs a detailed placement technique called Dual Objective Independent Set Matching (DOISM) [1] to improve both wirelength and external pins. DOISM is an extension of ISM which was used successfully in UTPlaceF [22] for the first time. The ISM algorithm uses a bipartite graph matching approach to improve only wirelength.

Figure 4.1 shows an example of an ISM bipartite graph. In this graph, there are two disjoint sets. The first set consists of logic blocks while the second set contains the current locations of logic blocks as well as some free spaces called White-Space. Each logic block in the first set is connected to all of the locations in the second set by a weighted edge. The weight of each edge is determined by the change in HPWL wirelength that would occur if the logic block is moved to the location. A minimum weight matching is performed to minimize the wirelength. The key concept in ISM is to ensure all the cells in the first set do not share any nets. Therefore, all cells can be moved without affecting the wirelength of any of the other cells in the set.

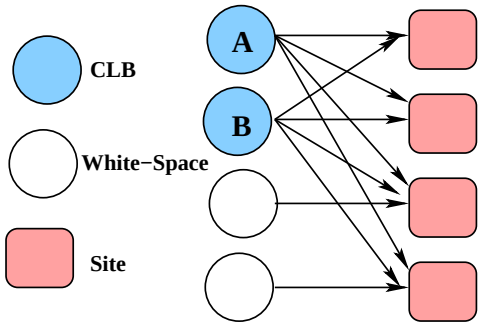


Figure 4.1: ISM Bipartite Graph [1]

DOISM leverages the idea of ISM, but it also employs this technique to reduce the number of external pins to improve the routability. An overview of the DOISM framework is presented in Figure 4.2. ISM minimizes wirelength by moving logic blocks within a specified window. DOISM uses the same algorithm but applies it to optimize different objectives. It alternates between optimizing wirelength and reducing external pins to improve both wirelength and routability. DOISM has an iterative flow. During each iteration, one objective is optimized while having a threshold on the amount of

degradation on the other objective.

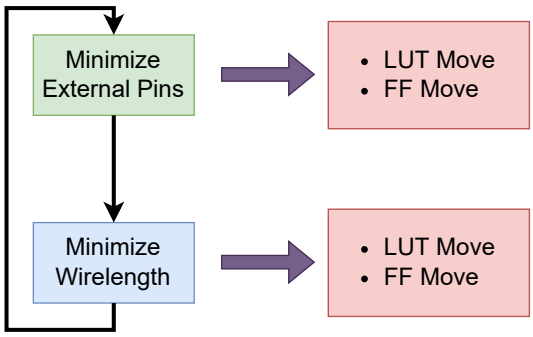


Figure 4.2: Dual-Objective ISM

4.1.1 The DOISM Flow

Algorithm 2 describes DOISM in more detail. The DOISM can optimize either wirelength or external pins by changing the weights in the independent set-matching graph. The main loop (lines 1-11), iterates over every node (LUTs, FFs) in the placement until the improvement drops below a threshold I_{th}. In each iteration, a node that has participated in DOISM the fewest number of times is selected and a window (local neighborhood) is created around that node (line 2). The objective (which is improving either wirelength or external pins) is defined by the static flow. This window includes all other nodes and white-spaces within a distance D_{is} of the chosen node. Using the nodes that are within the window, an independent set is generated (line 3). Then, each of these nodes and their locations on the FPGA is used to create a bipartite graph (line 4). The cost of the edge connecting each node to its location is calculated and added to the bipartite graph (lines 5-8). A min-cost bipartite matching is then performed using the Hungarian method (line 9). Consequently, the location of each node is updated (line 10). An important point

about the ISM actions is that a move is accepted if at least one objective is improved. Besides, the ISM action space includes actions that limit the amount of degradation on one objective while it optimizing the other objective. As we go through the ISM flow, the possible amount of degradation is gradually reduced to not allow the deterioration of the objectives too much at later iterations. This is also defined by the static flow provided in the GPlace3.0 [1] at the beginning of the detailed placement phase.

Algorithm 2 Dual-Objective Independent Set Matching (DOISM) [1]

Require: Maximum independent set size N_{is}, maximum independent set radius D_{is}, required improvement threshold I_{th}

while $improvement > I_{th}$ **do**
  Select a window W with a radius of D_{is} nodes
  $S \leftarrow$ **GenerateIndependentSet**(W)
  Add the nodes in S and their locations to bipartite graph g
  for each pair (n, l) with $n \in$ node set of g, $l \in$ location set of g **do**
    $cost \leftarrow$ **GetMovingCost**(n, l)
    Add the edge $(n, l, cost)$ to g
  end for
  Perform min-cost bipartite matching on g
  Update the location of each node in g
end while

4.2 Proposed Framework

Figure 4.3 shows an overview of our proposed framework that integrates the RL models into the detailed placement of GPlace3.0. In this framework, we propose to divide the FPGA chip into a grid of regions. The UltrasScale FPGAs have a width and height of 180 CLB, and 480 CLB, respectively. In the proposed framework, the FPGA is divided into regions with a size of 21 * 21 CLB leaving us with 184 regions. Partitioning the FPGA fabric to different regions enables the framework to capture the different characteristics of each region and dynamically adjust the optimization strategy. Our goal here is to

improve wirelength and external pins in local neighborhoods. Moreover, with this grid of regions methodology, we can provide the agent with more training samples after each iteration. The justification for using a grid size of 21 CLB is that this represents the same window size utilized in GPlace3.0 ISM. In the next step, for each region, several features are extracted. These features are used to describe the placement within each region. The extracted features are then fed into the RL agent. The agent would select an action for each distinct region based on its features. By executing an appropriate action, a new placement will be produced. The simulated environment provides the RL agent with different solutions so that the agent produces new actions. Once the placement solution has converged (i.e. there is no further improvement) the RL model terminates the optimization process.

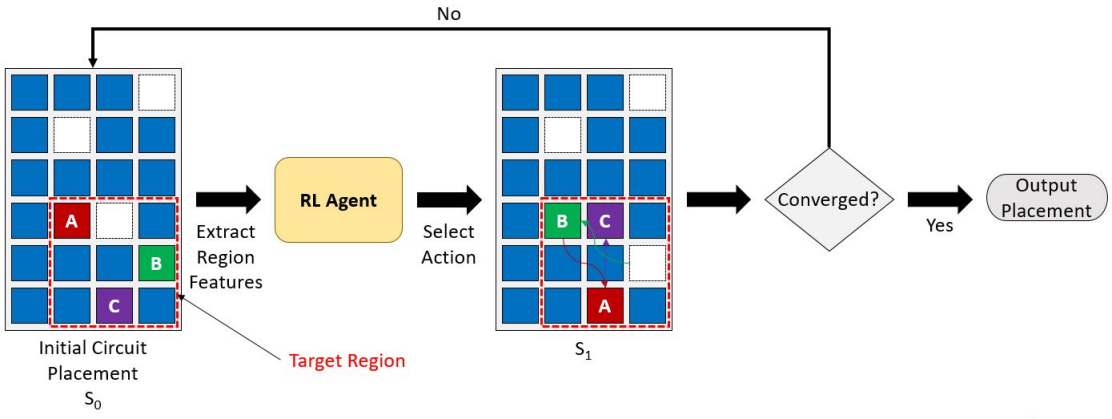


Figure 4.3: Proposed Framework

Algorithm 3 describes the RL-ISM algorithm in detail. The main loop (lines 1-24), iterates over every node (LUTs, FFs) in the placement until the improvement drops below an improvement threshold I_{th}. In each iteration, RL-ISM is performed across all of the grid regions (lines 2-22). For each region, placement features are extracted (line

3). These features are then passed to the RL agent and it will specify the next action for that region (line 4). An action counter is set to 0 (line5). It counts how many times the selected action is performed in that region. Then, the selected action is performed IpA times in that region before going to the next region (line 6-17). A random location is selected in that region and a window (local neighborhood) is created around that random location (line 7-8). Based on the action type selected by the RL agent, this window includes all nodes that have the same type and white-spaces within a distance D_{is}. Using the nodes that are within the window, an independent set is generated (line 9). Then, each of these nodes and their locations on the FPGA is used to create a bipartite graph (line 10). The cost of the edge connecting each node to its location is calculated and added to the bipartite graph (lines 11-14). A min-cost bipartite matching is then performed using the Hungarian method (line 15). The action counter is also updated (line16). Consequently, the location of each node is updated (line 18). The amount of improvement in both objectives is calculated and used as the reward signal for the RL agent's action (line 19). The new placement features are calculated to identify the next state(line 20). All of the transitions are saved in a replay memory (line 21). Finally, we use the replay memory to train our agent (line 23).

4.3 RL Components

A general overview of the RL framework is illustrated in Figure 4.4. The figure highlights three main components in the RL frameworks including the Environment, Agent, and Reward signal. The agent takes different actions as it interacts with the environment and in return, it receives an immediate reward for each action. The Agent's objective is to

Algorithm 3 RL ISM

Require: Maximum independent set size N_{is}, maximum independent set radius D_{is}, required improvement threshold I_{th}, Iterations per Action IpA

while $improvement > I_{th}$ **do**
  for each region $P \in$ grid **do**
    Extract features S of region P
    Action $A \leftarrow$ **RL-AgentGetAction**(S)
    $Num_{Actions} = 0$
    for $Num\_Actions < IpA$ **do**
      Select a random location X in region P
      Select a window W with a radius of D_{is} and X as the center
      $T \leftarrow$ **GenerateIndependentSet**(W) with respect to A
      Add the nodes in T and their locations to bipartite graph g
      for each pair (n, l) with $n \in$ node set of g, $l \in$ location set of g **do**
        $cost \leftarrow$ **GetMovingCost**(n, l)
        Add the edge $(n, l, cost)$ to g
      end for
      Perform min-cost bipartite matching on g
      $Num_{Actions} = Num_{Actions} + 1$
    end for
    Update the location of each node in g
    Calculate reward R of action A
    Extract new features S' of region P
    Save (S, A, R, S') transition into Replay Memory M
  end for
  RL-AgentTrain(M)
end while

maximize its expected cumulative reward. In the following subsections, we will delve into more details about each of these components in the detailed placement problem.

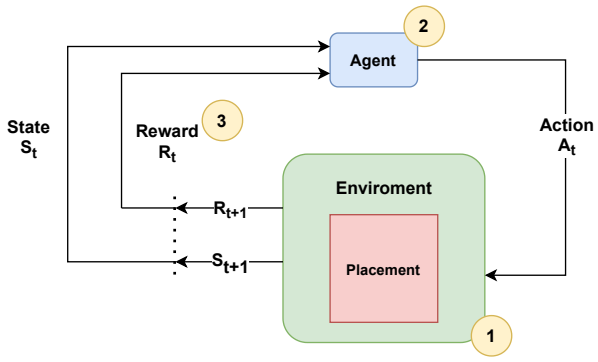


Figure 4.4: RL Framework

4.3.1 Agent

The RL agent is composed of two main components. The first component in the form of "Action Space" represents the actions that the agent can take in each state. The second component is the selection criteria used to select an action from the action space.

The action space in this problem is illustrated in Table 4.1. Each action consists of two parts. The first part is concerned with the optimization objective i.e. either wire-length or external pins. The second part relates to the type of logic block used to optimize the objective function. In this specific problem, logic types are either Look-Up-Table (LUT) or FlipFlop (FF). For example, LUT_WL translates to moving LUTs to optimize Wirelength. Therefore we will have 4 actions in total.

Objective / Cell Type	Wirelength	External Pins
LUT	LUT_WL	LUT_EP
FF	FF_WL	FF_EP

Table 4.1: Action Space

4.3.2 State Space

In any RL setup, the environment is usually represented with a set of states. Each state should be able to fully describe the current status of the environment. In this thesis, the state of the environment is represented by the current placement solution of the circuit netlist on the FPGA fabric. To identify different states, we capture several features of the placement in our target region. Our features are based on the features that were used in the work [39] described below.

- **Estimated Local HPWL:** This feature represents the estimated wirelength (HPWL) in each region. HPWL within each region is directly correlated with the wirelength of placement which we want to optimize in this problem.
- **Local External Pins:** This feature represents the number of external pins within each region. The total external pins of the placement are directly correlated with the external pins of each region.
- **LUT density:** This feature is calculated by dividing the number of utilized LUTs by the number of available LUTs in each region. This feature can assist the agent

to decide which cell type it should choose to optimize its objective.

- **FF density:** This feature is calculated by dividing the number of utilized FFs by the number of available FFs in each region. This feature can assist the agent to decide which cell type it should choose to optimize its objective.

- **Mean Wire Congestion:** This feature represents the mean wire congestion in each region. This feature is calculated using the work in [40]. The wire congestion is calculated by dividing the wirelength by the available routing channels. In other words, this feature can be another representation of the wirelength in each region.

- **Global Wirelength Improvement Rate:** This feature represents the wirelength improvement rate across the whole FPGA. The above-mentioned features cannot provide the agent with any information about the big picture of wirelength optimization. Moreover, this feature will change in a bigger range, despite the local features, which can help the agent to distinguish between different stages of the placement.

- **Global External Pins Improvement Rate:** This feature, similar to the Global Wirelength Improvement Rate, can aid the agent to capture the information related to external pins optimization across the whole FPGA and differentiate between different stages of the placement.

All features are normalized to be between 0 and 1 using the statistics of that feature during the baseline run. In each RL model, we used a subset of these features to describe our states. In the following sections, we describe the state features of each model.

4.3.3 Reward

WThe detailed placement is a multi-objective optimization problem in which both wire-length (HPWL) and external pins (EP) need to be optimized. After each action, we produce an immediate reward based on the amount of improvement. In this thesis, we have explored 2 different reward functions. Equation 4.1 present our first reward function. The reward associated with each action is equal to the linear combination of the normalized improvement of both objectives. The improvements of both objectives are negative, (i.e. decrease in WL, #pins). Therefore we multiply each by negative one to produce a positive reward.

Equation 4.2 illustrates the second reward function explored in this experiment. In this function, if both objectives are improved more than a predefined threshold, we use the same reward signal as Equation 4.1. However, if the amount of improvement in either of the objectives drops below the threshold, we generate a constant negative reward.

$$\text{R(s, a)} = -[\lambda(\Delta HPWL) + (1-\lambda)(\Delta EP)] \tag{4.1}$$

$$\text{R(s, a)} = \begin{cases} -[\lambda(\Delta HPWL) + (1-\lambda)(\Delta EP)] & \text{Otherwise} \\ -C & \text{if } \Delta HPWL < T_1 \ \& \ \Delta EP < T_2 \end{cases} \tag{4.2}$$

In the Tabular Q-Learning and Deep Q-Learning models, we only explore the reward function presented in 4.1. In the Advantage Actor-critic model, we investigate both of the reward functions.

4.4 Tabular Q-Learning (TQL) Model

The first model that was investigated is a tabular Q-learning model. This model uses a table, known as Q-table, to store state-action values. A Q-table for N states and M actions is presented in Figure 4.5. In this table Q(S, A) corresponds to the value of the state-action pair and it is used to estimate the quality of each action given a state. First, based on the values of the state features, we determine the state of each region. Next, given the current state, we find the value of each action using the Q-table. By using an action selection strategy, the best quality action is selected and performed and the Q-table is updated with respect to the reward that is received from the environment. In this model, we use the reward function introduced in Equation 4.1. Figure 4.6 presents an overview of this model. A description of the different modules of this model will also be presented.

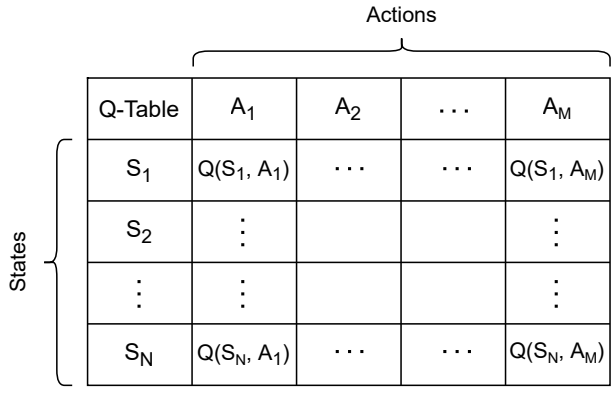


Figure 4.5: Q-table Example

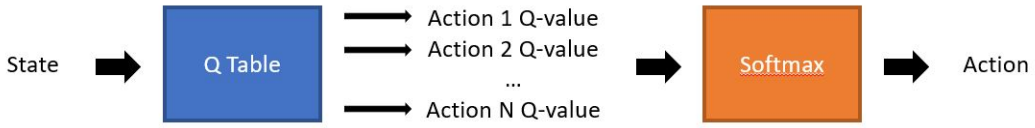


Figure 4.6: Q-Learning Model

4.4.1 State-Action Space

The tabular Q-Learning model can only be used in discrete state-action spaces. the action space here is the same as the action space described in section 4.3.1, therefore, four actions will be available in every state. For the state space, three features will be used to identify the state of a region. (1.) Mean Wire Congestion (2.) LUT Density (3.) FF density. To create a discrete state space, each of these features is divided into several bins. To be more specific, Mean Wire Congestion is divided into six bins while FF and LUT densities are divided into four bins. Therefore, in this model, we will have 96 states in total. This produces a table with 96 rows and 4 columns.

4.4.2 Softmax Action Selection Strategy

An Action selection strategy is crucial to establish a balance between exploration and exploitation of the solution space. In this model, a Softmax function is used to choose between actions. The softmax function takes a vector Z of K real numbers and normalizes it into a probability distribution consisting of K probabilities proportional to the exponentials of the input numbers whose total sums up to 1. Furthermore, the larger input components will correspond to larger probabilities which means the most valuable actions have the highest chance to be chosen. Equation 4.3 demonstrates the nature of

softmax function.

$$\text{Softmax}(x_i) = \frac{e^{x_i}}{\sum_j e^{x_j}} \tag{4.3}$$

In this strategy, the probability of an action to be selected is very dependent on the Q-value associated with that action. Therefore, to make sure that every action has a chance to be selected, we add a parameter, called Minimum Action Probability (MAP). This parameter ensures that the probability associated with each action does not drop below a specific threshold even if its Q-value is very small. This can help the RL agent to explore the environment and learn better about each action in different states. In this thesis, the Minimum Action Probability for all actions is set to 1%.

4.4.3 Training

In this model, the RL agent is trained online over the course of a single episode (benchmark), and all the information is discarded with starting the next episode. For each benchmark, the RL agent is trained separately. Throughout the episode, in each iteration, 184 samples are generated which are associated with the transition in each region of the placement. Then, these samples are used to train the RL agent. The training process updates the values in the Q-table after taking each action and receives its associated reward from the environment.

4.4.4 Model Parameters

Table 4.2 shows the parameters associated with the Tabular Q-Learning model. Below, we describe each parameter in more detail.

- **Learning Rate (α):** Learning Rate defines the rate at which new data modify the Q-values in the training process. Setting the learning rate to 0 means that the Q-values are never updated. While setting it to higher values like 1 means that the learning should occur as quickly as possible.

- **Discount Factor (γ):** Defines how much the RL agent cares about the immediate reward versus the rewards in the distant future. Higher values mean that the RL agent will value the future and give higher rewards for actions that take the future into account. With the lower discount factor, the RL agent will care more about the short-term rewards in a greedy manner.

- **Minimum Action Probability:** This parameter makes sure that the probability associated with selecting an action does not become very small. Higher numbers will help the agent to explore the environment better in different states. However, it can lead to poor quality of results since the more valuable actions will be selected fewer times.

- **Iterations per Action (IpA):** Defines the number of times that the selected action will be performed before getting back to the RL agent for the next action.

- **Objective Weights (λ):** This parameter determines the importance of each objective by assigning a weight to each. Setting this parameter to 0 means that we only care about the external pins while λ equal to 1 means the wirelength is the sole objective.

- **Batch Size:** It is the number of transitions samples used to train the RL agent in each iteration.

Parameter	Acceptable Range
Learning Rate	0 - 1
Discount Factor	0 - 1
Minimum Action Probability	0% - $(100/N_{actions})\%$
Iterations per Action	1 - 50
λ (Objective Weights)	0 - 1
Batch Size	184

Table 4.2: Tabular Q-Learning Model Parameters.

4.5 Deep Q-Learning Model (DQL)

One of the main limitations of the Tabular Q-learning model is that it best suits problems with a small discrete state space. However, in the detailed placement problem, there can exist many possible placement solutions which are best represented by continuous features. A small discrete state space will often fail to capture the gradual changes in the placement. To address this issue, we propose to investigate a Deep Q-Learning model where the states are defined by a set of continuous features.

Deep Q-Learning models employ Artificial Neural Networks (ANN) as a function estimator to estimate the Q-values instead of storing them in a Q-table. An overview of the Deep Q-Learning model is depicted in Figure 4.7. In this model, we pass the state features of each region as input to an ANN. The ANN estimates the value of each action based on the state features. These values are then passed to an action selection function to find the most suitable action for that region. The selected action is performed and a

reward is generated by the environment transitioning to a new state (new state features). Then, the entire transition including state, action, reward, and next state (s_t, a_t, r_t, s_{t+1}) is stored in a replay buffer. This process is repeated for all the regions within the FPGA grid. For the DQL model, the reward function described in Equation 4.1 is used. In the next few subsections, different parts of the DQL model will be explained in detail.

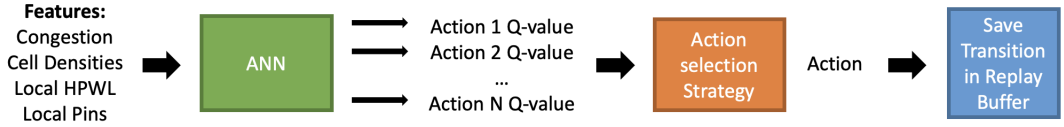


Figure 4.7: Deep Q-Learning Model

4.5.1 The State-Action Space

In this model, the action space is the same as the action space presented in section 4.2.1.2. For the state space, five features are proposed to be used as the state features vector to identify the state of each region at each time of the placement process. The state features are as follows (1.) Mean Wire Congestion (2.) LUT Density (3.) FF density (4.) Local HPWL (5.) Local External Pins. The term ”local” here indicates that the wirelength (HPWL) and External pins are calculated within a region.

4.5.2 The ANN Architecture

In the DQL model, the agent relies on an Artificial Neural Network to estimate the state-action values. The input to this network is the state features describing the current state of the environment. The output of this network is the Q-value associated with each action given the current state. The architecture of the ANN used in this thesis is illustrated

in Figure 4.8. The input layer of this ANN has five inputs which are associated with the five state features described earlier in the previous section. The ANN has a single hidden layer with fifty neurons. This layer uses a Relu function as the activation function. Finally, the output layer of the ANN has four outputs with a linear activation function corresponding to one of our four actions. The output value of each neuron in the output layer is the estimated value of each action given the state features vector.

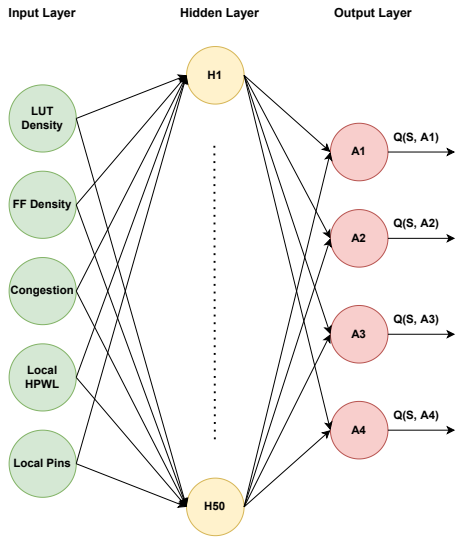


Figure 4.8: ANN Architecture

4.5.3 Action Selection Strategies

In this model, we have explored two different action selection strategies. Below, we describe each strategy in more detail highlighting the advantages and disadvantages.

4.5.3.1 Epsilon-Greedy Selection Strategy

ϵ-Greedy is an exploration strategy widely used in reinforcement learning problems to handle exploration-exploitation trade-off by choosing between exploration and exploitation randomly. In this strategy, ϵ refers to the probability of choosing to explore the environment while 1-ϵ is the probability of choosing a greedy action to maximize the cumulative reward. Equation 4.4 has illustrated how an ϵ-greedy function looks like.

$$\text{Action}(a) = \begin{cases} \text{random action (a)} & \text{with probability } \epsilon \\ \arg\max_a Q(s,a) & \text{with probability } 1-\epsilon \end{cases} \tag{4.4}$$

ϵ usually starts with higher values to make sure that the RL agent explores the environment and learns about the value of each action in different states. As we go through the episode, the ϵ is reduced with a decay rate and actions with higher rewards will be chosen more often. Using this strategy, the actions associated with higher rewards in the early stages of the episode will be more likely to be selected later while other actions have a very small chance to be selected. Consequently, this strategy will lead to a deterministic behavior policy when ϵ is small enough. The deterministic policies can lead to a sub-optimal solution in the problems we need a combination of different actions to achieve the optimal solution. Another disadvantage of this method is that it only explores the environment at the early stages of the episode while actions may have different values in the later stage of the episode.

4.5.3.2 Softmax Selection Strategy

The Epsilon-greedy strategy is a popular means of balancing exploration and exploitation in the reinforcement learning problems as it has shown effectiveness in a wide range of problems. However, a problem with this strategy is that it chooses equally among all the actions when it explores the environment. In other words, the worst-appearing action has the same chance as the next-to-best action. Another problem, as we stated earlier, is that when the agent decides to exploit what it has learned, it always chooses the action with the highest Q-value. This can be undesirable when there are actions with the Q-value very close to the optimal action.

A solution to this problem is to vary the action probabilities as a graded function of the estimated value. The action with the highest will still have the highest chance to be selected, but all the other actions are ranked according to their estimated Q-values. This approach is called the softmax action selection strategy. We have already discussed the functionality of this approach in section 4.4.2. The DQL model architecture using the softmax strategy is illustrated in Figure 4.9.

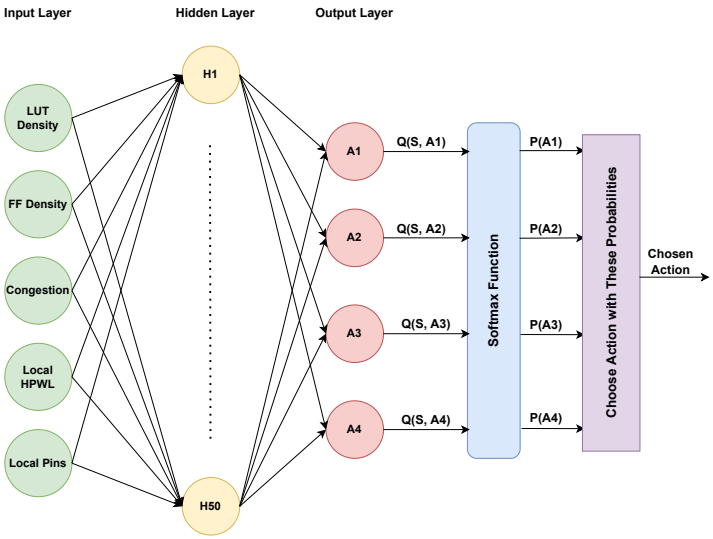


Figure 4.9: DQL Architecture with Softmax

One problem with the softmax selection strategy is the following: If there is a large enough error in Q-value estimates, it can get stuck as the exploration could heavily favor a current best value estimate. For instance, if one estimate is accurate and relatively high, but another estimate is much lower but in reality would be a good action choice, then the softmax probabilities to resample the bad estimate will be very low and it could take a very long time to fix.

A more serious problem is that the differences between optimal and non-optimal Q value estimates could be at any scale, maybe just a 0.1 difference in value, or maybe 100 or more. This makes softmax a poor choice as it might suggest a near-random exploration policy in one problem, and a near deterministic policy in another, irrespective of what exploration might be useful at the current stage of learning.

4.5.4 Training

In the traditional tabular Q-learning approach, a memory in the form of a Q-table is built to store Q-values for all possible combinations of states and actions. However, this method can only be used in environments with discrete state-action space. To overcome this limitation, an ANN is proposed to approximate Q-values. With the DQL model, we generalize the approximation of the Q-value function rather than remembering the solutions. However, employing neural networks and training them adds more challenges to the reinforcement learning problems. Fortunately, there are some tricks that help us to overcome these challenges. Below, we introduce two major challenges in training DQL models and their associated solutions.

4.5.4.1 Replay Buffer

In the DQL model, we try to estimate a nonlinear function, Q(S, A), with a neural network. One of the most fundamental requirements for training such networks is that the training data should be independent and identically distributed. However, in the naive deep Q-learning models, the network is trained with a sequence of experience tuples which are highly correlated making the model oscillate by the effect of this correlation.

To prevent this oscillation, it has been proposed to store previous experiences in a replay buffer and sample training data from it, instead of using the most recent experience. The replay buffer contains a collection of transition experience tuples (State, Action, Reward, Next State) which are added to the replay buffer as the RL agent interacts with the environment.

In this thesis, we use a fixed size (10K tuples) replay buffer in which the new data

is added to the end of the buffer pushing the oldest experience out of it. For training, we use a batch of previous experiences sampled from the buffer to update the Q-network rather than using just the most recent experience. This should help in breaking the correlation between consecutive transitions. Moreover, this buffer allows the reuse of past experiences to avoid catastrophic forgetting. This is a common problem training ANNs when prior knowledge is unlikely to be kept intact. In this thesis, a batch of 736 samples is used to train the agent in each iteration.

4.5.4.2 Target Network

Equation 4.5 below illustrates how the Q-values are updated generally in the Q-learning models. The TQL and DQL models use the same equation for learning the optimal Q-values. However, the main difference between them is that the exact value function (Q-table) is replaced with a function approximator in the latter one (DQL). As a result, in the DQL model, both the current value, $Q(s_t, a_t)$, and the target value, $Q(s_{t+1}, a_{t+1})$ are calculated using the same network. Therefore, with updating the network weights, the target value will also change in the same direction. This will make the training process very unstable

$$Q(s_t, a_t) = Q(s_t, a_t) + \alpha[r_t + \gamma \max_{a_{t+1}} Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)] \tag{4.5}$$

To overcome this problem, it is proposed to use a target network for calculating the target $Q(s_{t+1}, a_{t+1})$ values and use these values for training the main Q-network. The target network is basically a copy of the main Q-network with the same architecture. The weights of the target network are not trained but periodically synchronized with the

parameters of the main Q-network.

In this thesis, we use the same ANN architecture illustrated in Figure 4.8 as our target network. Weights of this network are replaced with the weights of the main Q-network every k step. In our case, the k parameter is known as Target Frequency Update and it is set to 1000. It means the target network's weights are updated when the main Q-network has been trained 1000 iterations.

The proposed agent is trained throughout a single episode, and the weight of the Q-network and replay buffer transitions are discarded with starting the next episode.

4.5.5 Model Parameters

Table 4.3 shows the parameters associated with the Deep Q-Learning model. We already covered most of these parameters in section 4.4.4. Below, we describe the parameters introduced in the DQL model.

- **Target Update Frequency:** This parameter controls how often the weights of the target network are synchronized with the weights of the Q-network.

- **Epsilon:** This parameter controls the amount of exploration versus exploitation in ϵ-greedy strategy. It starts with 100 which means all the actions have the same chance to be selected. It gradually decays to 1 which means 1% of the time a random action is selected and 99% of the time, the most rewarded action is selected.

- **Epsilon Decay Rate:** It defines the rate that the Epsilon parameter decays during an episode. This value is deducted from the Epsilon after each action selection.

Parameter	Acceptable Range
Learning Rate	0 - 1
Discount Factor	0 - 1
Target Update Frequency	1000
Epsilon	Decays from 100% to 1%
Epsilon Decay Rate	0.0000099
Iterations per Action	1 - 50
λ (Objective Weights)	0 - 1
Batch Size	736

Table 4.3: Deep Q-Learning Model Parameters.

4.6 Advantage Actor-Critic (A2C) Model

The Advantage Actor-Critic [41] is a variation of the actor-critic model where the critic model learns the Advantage value (A-value) function A(s, a), instead of Q-values. Equation 4.6 shows how the Q-value is decomposed into a V-value, and A-value. The state value function calculates how good is to be at state s, while the advantage function calculates how the goodness of an action is compared to the others at a given state s. To calculate the advantage function, two functions are required. The action-value function Q(s, a), and the state value function V(s). Fortunately, we can use the TD error as a good estimator of the advantage function and rewrite the equation 4.6 as equation 4.7. Therefore, the update rule of the actor will change accordingly, equation 4.8. In this equation, the Q-value is replaced by A-value. If the advantage is positive, the actor will update its

parameters θ in a way that it is more likely to choose action a_t in state s_t. Otherwise, the probability of taking action a_t in state s_t will be decreased. The main benefit of the advantage function is that it reduces the high variance of policy networks and stabilizes the model.

$$A(s_t, a_t) = Q(s_t, a_t) - V(s_t) \tag{4.6}$$

$$A(s_t, a_t) = r + \gamma V(s_{t+1}) - V(s_t) \tag{4.7}$$

$$\theta_{t+1} = \theta_{t+1} + \alpha A(s_t, a_t) \nabla_\theta log \pi_\theta(a_t|s_t) \tag{4.8}$$

In this model, we explore both reward functions presented in section 4.3.3. Besides, both the actor and critic models are parameterized using artificial neural networks. The architecture of these networks is described in subsection 4.6.2.

4.6.1 State-Action Space

The action space of the A2C model is similar is to the action space presented in section 4.3.1. For the state space, six features are proposed to identify the state of each region at different times during the placement process. The state features are as follows: (1.) LUT Density (2.) FF Density (3.) Local HPWL (4.) Local External Pins (5.) Global Wirelength Improvement Rate, (6.) Global External Pins Improvement Rate. The global HPWL and global External Pins improvement rates were added to assist the agent to distinguish between different stages of the placement. Also, these new features can provide

the agent with global information related to the optimization of objectives. Moreover, the Mean Wire Congestion is eliminated in this model since the experiments from previous models showed that this feature changes in a very small range within each region which cannot be captured by the agent. On the other hand, calculating this feature is very expensive in terms of CPU time. Therefore, the runtime of this model can be improved significantly while the same QoR obtained.

4.6.2 Actor-Critic Networks Architecture

Advantage Actor-Critic (A2C) models need to parameterize 2 functions: (1.) Policy function (Actor), (2.) State Value function (Critic). These two functions can be parameterized using 2 artificial neural networks which can share some parameters or be completely independent. Sharing some parameters has advantages and disadvantages. On the one hand, sharing parameters between two networks reduces the total number of trainable parameters which increases the sample efficiency of the model. Especially in Actor-Critic methods, the actor is reinforced by a learned critic. Therefore, using separate networks, the actor may not learn anything until the critic learned enough. However, with shared parameters, the actor can benefit from what has been learned by the critic thus improving the sample efficiency. Additionally, learning in both policy and value networks is associated with learning how to cluster similar states together. Therefore, both networks can benefit from sharing the lower-level learning related to the state space.

On the other hand, sharing parameters between two networks tend to make the learning process unstable. This is because two gradients are now being back-propagated through the network simultaneously. The two gradients may have different scales which need to be balanced to have a stable learning process.

In this thesis, a shared architecture for the Advantage Actor-Critic (A2C) model is proposed. An overview of this architecture is depicted in Figure 4.10. The A2C model uses the same base network with two output heads. One will produce a probability distribution over the actions for the current state and the other will calculate the value of that state. The input layer of the base network has six inputs which are the state features. Fifty neurons are deployed in the hidden layer with Relu activation functions. The actor's output layer has four outputs which are the 4 possible actions and the output of each neuron is the probability of the associated action to be selected. A softmax function is used as the activation function in this layer to generate the probability distributions over the action space. The critic has a single output with a linear activation function that will determine the value of the current state.

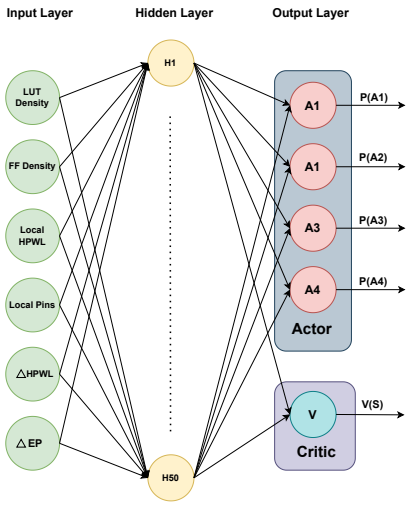


Figure 4.10: Actor-Critic Networks Architecture

4.6.3 Actor-Network: ”Selection Strategy”

The policy network, (Actor), outputs a probability distribution corresponding to each action. Actions from this probability distribution are sampled with respect to each action's probability.

The difference between a policy network used in this model and the value network used in the DQL model is that the Q-values produced by the valued network have an inherent meaning based on summed rewards. While in the policy network, the probability of each action indicates the preference to choose that action given a state.

4.6.4 Training

In the DQL model, a replay buffer was proposed to be used to break the correlation of consecutive inputs. However, this technique cannot be used in the A2C model since it is an on-policy algorithm. The on-policy means that we are learning the value function for the policy we are following right now. However, if we sample past experiences from the replay buffer, we are using the information that is gathered by following another policy. Therefore, in the A2C model, we train the RL agent at every time step with the samples that are generated in that time step rather than sampling past experiences from the replay buffer.

When TD learning is combined with function approximation, updating the value at one state creates a risk of inappropriately changing the values of other states, including the state being bootstrapped upon. This is not a concern when the agent updates the values used for bootstrapping as often as they are used (Sutton et al., 2016). However, if the agent is learning off-policy, it might not update these bootstrap values sufficiently

often. This can create harmful learning dynamics that can lead to divergence of the function parameters (Sutton, 1995; Baird, 1995; Tsitsiklis and Van Roy, 1997). The combination of function approximation, off-policy learning, and bootstrapping has been called the deadly triad due to this possibility of divergence (Sutton and Barto, 2018). To prevent model parameters from divergence, we employed a target network that is periodically synchronized with the main Q-network in the DQL model. However, the A2C model is on-policy, and therefore the target network is not needed for this model.

In this model, we propose to train the RL agent in two ways. First, the agent is trained online over the course of a single episode. Besides, for the different benchmarks, the agent is trained separately. Then, to further improve the learning process in this model, we propose to train the RL agent throughout several episodes on the same benchmark. After finishing each episode, all the networks' parameters are saved and then, with starting the new episode (same benchmark), these parameters are loaded into the RL model. In this model, we use a batch of 184 samples generated at each time step.

4.6.5 Model Parameters

Table 4.4 shows the parameter associated with the A2C model. We already covered these parameters in the previous RL models.

Parameter	Acceptable Range
Actor Learning Rate	0 - 1
Critic Learning Rate	0 - 1
Discount Factor	0 - 1
Iterations per Action	1 - 50
λ (Objective Weights)	30
Batch Size	184

Table 4.4: Advantage Actor-Critic Model Parameters.

4.7 Models Comparison

In this section, the different models investigated are compared. Tables 4.5 indicates the main similarities and differences between the three models introduced in this chapter. All three models are Model-Free Reinforcement Learning algorithms that use TD learning to train their agents. The TQL and DQN model are off-policy values-based models while the A2C model is an on-policy policy-based RL method.

TQL	DQL	A2C
Model-Free	Model-Free	Model-Free
Value-Based	Value-Based	Policy-Based
Off-policy	Off-Policy	On-policy
TD Learning	TD Learning	TD Learning

Table 4.5: Models Comparison.

4.8 Summary

In this chapter, we introduced the proposed framework in the thesis. Moreover, we presented all the different models investigated along with the possible drawbacks and limitations of each model. In the next chapter, we present the results produced by each model and analyze the results carefully.

Chapter 5

Results

In this chapter, we first introduce the experimental setup used to perform our experiments, section 5.1. Then, we present the results of the Tabular Q-Learning model, Deep Q-Learning Model, and Advantage Actor-Critic model in sections 5.2, 5.3, and 5.4, respectively. Additionally, we discuss the results of each model, along with the model's limitations. Finally, in section 5.5, we compare all of the models to each other and to GPlace3.0 ISM.

5.1 Experimental Setup

All of the RL models were developed using the Keras Deep Learning Library in python [42]. The analytic-placer that was used in this thesis, Gplace3.0 [1], is implemented using the C programming language and compiled using gcc 4.4 (Red Hat 4.4.7-18) compiler. All experiments are performed on a Linux machine (CentOS release 6.9) running on an Intel (Xeon CPU E3-1270 V2 @ 3.50GHz) processor.

5.2 Tabular Q-Learning Model (TQL) Results

In this section, we present the results of the tabular Q-learning model. We first introduce the initial results of the model. Then, we describe the different experiments that have been done to explore the effect of each parameter on the model's performance. Table 5.1 shows the model's parameters as well as their associated values during this experiment.

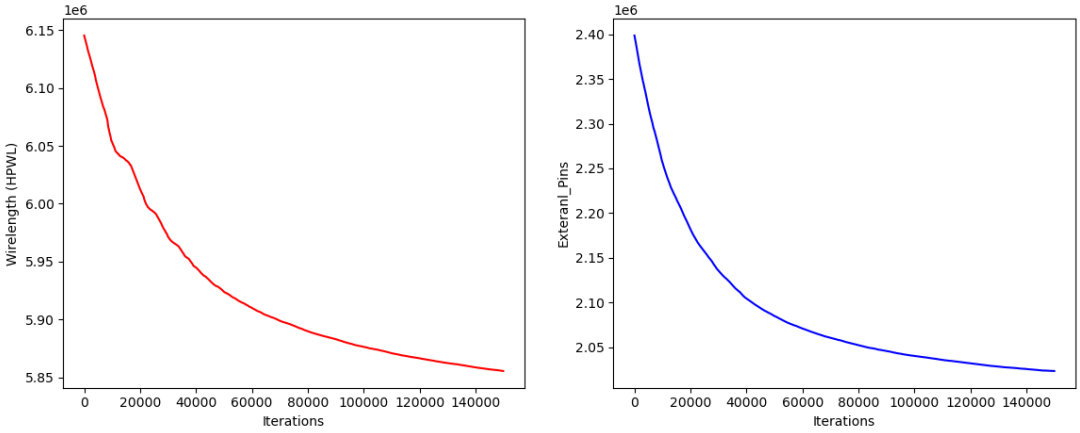
Parameter	Value
Learning Rate	0.01
Discount Factor	0.5
Minimum Action Probability	1%
Iterations per Action	1
λ (Objective Weights)	0.5

Table 5.1: Tabular Q-Learning Model Parameters.

Table 5.2 shows the average improvement in terms of wirelength and external pins over the 12 ISPD benchmark suite. This table indicates that the tabular Q-learning model was able to obtain 3.11% improvement in wirelength as well as 20.7% improvement in external pins. Figure 5.1 presents how the objectives are improving throughout the episode, optimizing FPGA-10. Figure 5.1a shows that the wirelength improvement rate decreases in the early stages. The reason behind this is that the external pin moves can deteriorate wirelength and reduce the improvement rate. Figure 5.1b shows the external pin improvement throughout the episode. External pins have a smoother improvement rate since wirelength moves are also helpful for external pins optimization.

Objectives	WL%	EP%	Total
Average Improvement	3.11	20.7	23.81

Table 5.2: Tabular Q-Learning Model Performance.



(a) Wirelength (b) External Pins

Figure 5.1: Objectives Improvement During Episode

Figure 5.2 illustrates the action selected by the agent, optimizing FPGA-10. At the beginning of the episode, the external pin moves (LUT_Pin, FF_Pin) are selected more frequently. That is why the wirelength improvement rate is reduced at the beginning of the episode. In this model, we are using a softmax strategy to choose between different actions. This strategy is very dependent on the Q-values. It means that if the Q-values are very close to each other, e.g. 0.1 difference in value, the RL agent will follow a near-random policy. However, if the Q-values are very different, the probability associated with the action that has the highest Q-value will dominate other actions' probability

leading to a deterministic policy.

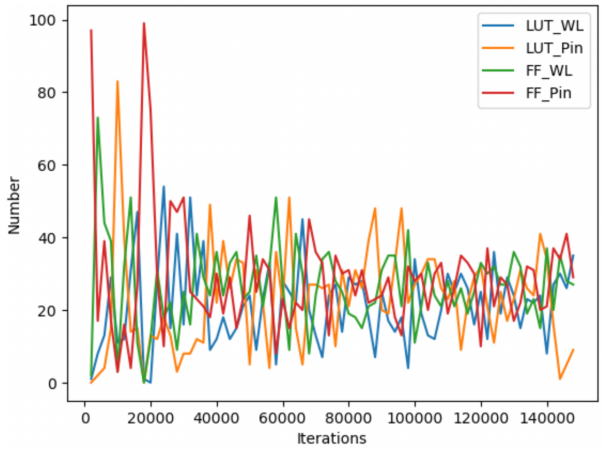


Figure 5.2: TQL Actions with Softmax Strategy

Below, we explore additional experiments to see the effect of individual parameters on the model's performance.

5.2.1 Effect of Iterations per Action (IpA):

In this experiment, the objective is to determine how the number of iterations associated with a specific action affects QoR. After calculating the state features of each grid across the FPGA and sending those features to the RL agent, the agent determines the best action for each grid with respect to the state of the grid. Iteration per Action is associated with the number of times that the chosen action is performed within the grid before getting back to the RL agent for the next move. The intuition behind this experiment is that during the detailed placement stage, we are making small changes to the placement within each grid. These small moves cannot change the state of the grids significantly and, therefore, the RL agent is more likely to choose the same action in the next iteration.

To have a better understanding of the role of IpA, we carry out this experiment with different numbers of IpA, including 1, 10, 20, 30, 40, and 50, while other parameters are fixed. Table 5.1 shows other parameters along with their associated value during this experiment. Moreover, we are using the tabular Q-Learning Model introduced in chapter 4 to train our RL agent. To evaluate this experiment, we used ISPD benchmark suite which includes 12 benchmarks with a wide range of features.

Table 5.3 shows how the objectives are affected by the IpA. The average improvement of wirelength and external pins associated with each IpA is presented. This table indicates that with increasing IpA from 1 to 20, the wirelength and external pins have improved 22%, and 27%, respectively. This is because, with increasing IpA, the variance of the reward signal is reduced significantly. As mentioned in section 4.1.1, each move is accepted only at least one of the objectives is improved, or nothing will move. Therefore, lots of actions will be associated with a reward equal to 0. With IpA equal to 1, the high variance of reward signal gets the RL agent into trouble learning the goodness of each action in a specific state. However, with higher IpAs, the reward variance is reduced effectively and can help the agent to learn more meaningfully.

On the other hand, Figure 5.5 also shows that with increasing IpA to 30 and more, the objectives start to deteriorate. This is because with increasing IpA, we will provide the model with fewer transition samples which can adversely affect the learning process and performance of the model. Besides, optimizing an objective too frequently can be harmful to the other objective.

IpA	1	10	20	30	40	50
WL %	3.11	3.65	3.8	3.51	2.83	-2.24
EP %	20.7	26.32	26.35	24.84	21.65	18.65
Total	23.81	29.97	30.15	28.35	24.48	16.41

Table 5.3: Effect of IpA in Tabular Q-Learning Model

5.2.2 Effect of Learning Rate:

In this experiment, our objective is to explore the effect of Learning Rate (α). Learning rate determines the speed at which the model learns. Setting the learning rate too high will make the model jump over minima, while a very low learning rate will either take too long to converge or get stuck in an undesirable local minimum.

To find a good trade-off between large and small learning rates in this problem, we repeated the same experiment using different learning rates ranging from 0.001 to 1. Besides, the IpA is set to 20 which has proven to improve the solution quality. Except for the IpA and learning rate, the rest of the parameters in Table 5.1 remain the same. To evaluate this experiment, we used the 12 benchmarks from ISPD benchmark suite.

Table 5.4 presents the average improvement in the objectives for 12 benchmarks. The results show that the tabular Q-learning model can generally obtain better QoR using smaller learning rates. However, setting it too low like 0.001 can lead to slightly worse results. This mainly happens for the smaller benchmarks since the episode length is shorter and we do not provide the agent with enough transition samples which adversely affect the solution quality.

α	0.001	0.01	0.1	0.5	0.9	1
WL %	3.7	3.8	3.51	3.34	2.83	2.74
EP %	25.8	26.35	24.84	24.53	21.33	21.17

Table 5.4: Effect of LR in Tabular Q-Learning Model

5.2.3 Effect of Discount Factor:

The last experiment for the tabular Q-learning model is investigating the effect of the Discount Factor (γ). Table 5.5 illustrates the effectiveness of different γ values. The discount factor determines the importance of future rewards versus immediate rewards. A discount factor equal to 0 means that the RL agent will only consider the immediate rewards similar to a greedy approach. In this case, the Q-values are representing the immediate reward. While setting the discount factor to a higher number will result in the Q-values representing the cumulative discounted future reward the RL agent expects to receive.

To see how the agent's performance is influenced by the discount factor, we repeat the same experiment a while different γ value is assigned. We change the value of γ from 0 to 1. The rest of the parameters are shown in Table 5.1, except for the IpA which is set to 20.

The result of this experiment is presented in Table 5.5. This table shows the average performance over the 12 ISPD benchmarks. The result shows the RL agent has better performance when the discount factor is set to the lower values. However, with increasing the γ to values more than 0.95, the agent's performance starts to deteriorate.

This shows that the agent has trouble accurately estimating the future rewards and this will mislead the agent during the episode. This is because the state features used in this model do not change significantly and meaningfully in different stages of placement leading to very slow state transitions. Therefore, one state-action that was associated with a high reward at the beginning of the episode where there are lots of room for improvement is expected to be highly rewarded in the later stages too. However, in practice, the reward associated with these state-action pairs is significantly reduced but the RL agent is not able to capture that. Second, the agent has trouble learning the effect of individual actions.

γ	0	0.1	0.5	0.9	0.99	1
WL %	3.73	3.74	3.74	3.51	2.2	1.96
EP %	26.3	26.36	26.35	24.84	17.45	16.11

Table 5.5: Effect of γ in Tabular Q-Learning Model

5.2.4 Model Evaluation

Conceptually, the tabular Q-learning model is one of the fundamentals for understanding reinforcement learning models. Moreover, it has the most straightforward implementation compared to other RL models. Another advantage of tabular Q-learning is that it can compare the expected reward of the available actions without requiring a model of the environment. On the other hand, one of the limitations of the tabular Q-learning is that it fits problems with discrete state-action space. This is problematic in the detailed placement problem, where the state features change slowly. In the detailed placement

stage, the placement solution is modified gradually resulting in small and slow changes in the state features. Furthermore, the Q-learning methods learn deterministic policies, while in lots of problems, a mixed strategy is needed to reach the optimal solution. Another disadvantage of this implementation is the sample inefficiency. In each iteration, samples are used to train the agent, and then all of them are discarded making the agent forgetful about the past experiences.

5.3 Deep Q-Learning Model (DQL) Results

In this section, we introduce the results of the Deep Q-Learning model. We first present the initial results of the model. Then, we describe different experiments that have been performed to explore the effect of different parameters on the model's performance. Table 5.6 illustrates the model's parameter and their correlated values during the initial implementation of the DQL model.

Parameter	Value
Learning Rate	0.01
Discount Factor	0.95
Target Update Frequency	1000
Epsilon	Decays from 100% to 1%
Epsilon Decay Rate	0.000099
Iterations per Action	1 - 50
λ (Objective Weights)	0.5

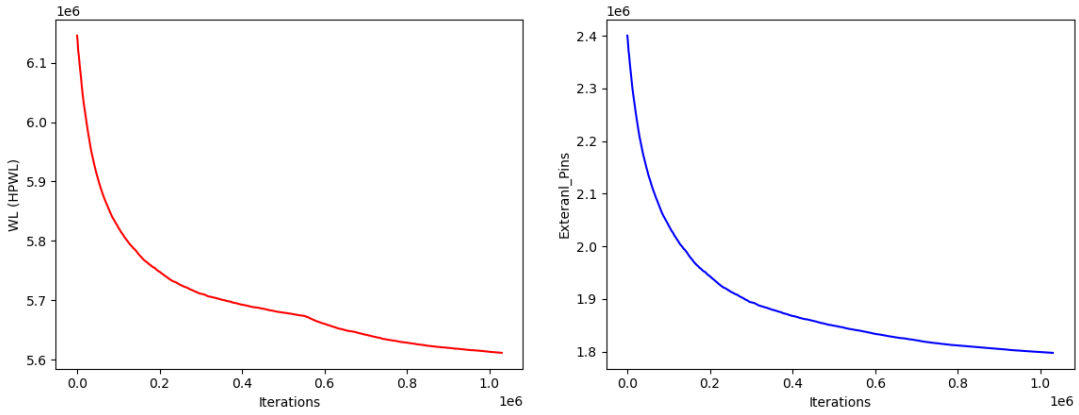
Table 5.6: Epsilon-Greedy Parameters

Table 5.7 presents the average improvement of the objectives over the 12 ISPD benchmarks. The DQL model was able to improve wirelength and external pins 5.3%, and 22.5%, respectively.

Objectives	WL%	EP%	Total
Average Improvement	5.3	22.5	27.8

Table 5.7: Deep Q-Learning Model Performance.

Figure 5.3 illustrates how the objectives are changing during the episode, optimizing FPGA-10. Figure 5.3a shows wirelength throughout the episode, while Figure 5.3b shows the number of external pins. The actions selected by the RL agent are depicted in Figure 5.4. Here we are using ϵ-greedy strategy as our action selection function. With this strategy, all the actions have the same chance to be selected at the early stage of the episode. However, ϵ is decayed gradually as we go through the placement and the agent is more likely to select the action with the highest Q-value. At some point, when the ϵ is small enough, the agent will always choose the most rewarded action resulting in a deterministic policy. Figure 5.4 shows that the DQL model sticks to the actions that are optimizing wirelength and completely ignores the actions associated with external pins. The reason behind this is that the moves that optimize wirelength are also improving external pins but external pin moves usually deteriorate wirelength. As a result, the actions that optimize wirelength are more rewarded and the agent will stick to those actions when exploration is finished.



(a) Wirelength (b) External Pins

Figure 5.3: Objectives Improvement During Episode

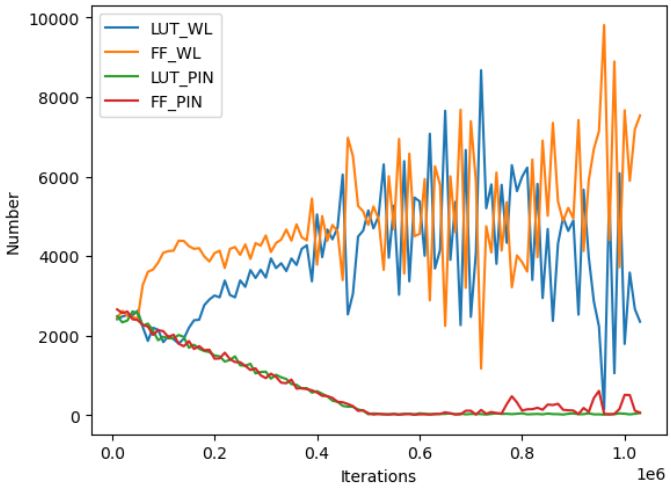


Figure 5.4: DQL Actions with Epsilon-Greedy Strategy

5.3.1 Effect of Iteration per Action (IpA):

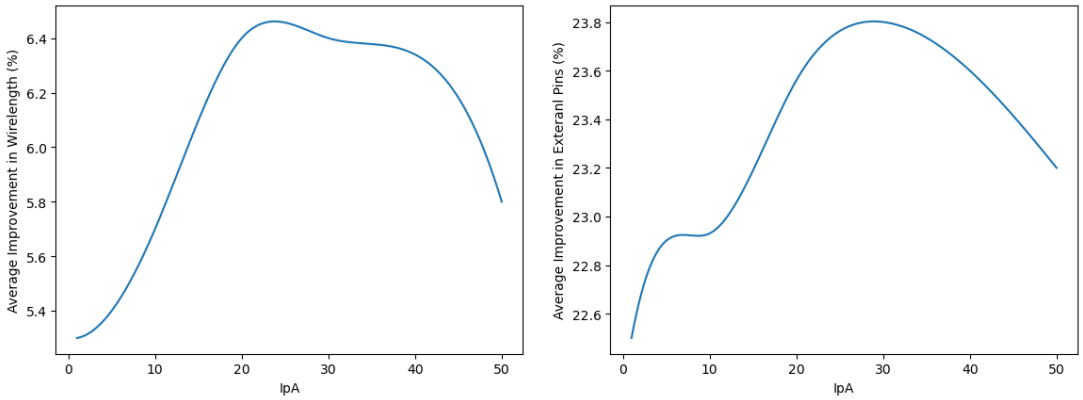
In this experiment, the objective is to determine how the number of iterations associated with a specific action affects QoR and runtime. We already explained this experiment in section 5.2.1. Table 5.6 shows other parameters along with their associated value during this experiment. Moreover, we are using the Deep Q-Learning Model with an ϵ-greedy strategy. To evaluate this experiment, we used the ISPD benchmark suite which includes 12 benchmarks with a wide range of features.

Effect on Objectives: Table 5.8 shows how the objectives are affected by the number of IpA. In this table, the average improvement in wirelength and external pins associated with each IpA is presented. Figure 5.5, shows how the objectives are changing with increasing IpA from 1 to 50. Figure 5.5a shows that increasing IpA from 1 to 30 helps to improve wirelength. External pins also start to improve when IpA is changing from 1 to higher numbers. This happens because with increasing the number of IpA, the variance of reward signal is reduced substantially.

On the other hand, Figure 5.5 also shows that with increasing IpA to 40 and 50, the objectives start to deteriorate. Since with increasing IpA, we will have fewer samples which will hurt the learning process. Besides, executing an optimization move too frequently can deteriorate the other objective.

IpA	1	10	20	30	40	50
WL %	5.3	5.7	6.4	6.4	6.34	5.8
EP %	22.5	22.93	23.56	23.8	23.6	23.17
Total	27.8	28.63	29.96	30.2	29.94	28.97

Table 5.8: Effect of IpA on Objectives in DQN Model.



(a) Wirelength (b) External Pins

Figure 5.5: Effect of IpA on Objectives in DQL Model

Effect on Runtime: In the proposed RL framework, all the state features need to be calculated when the RL agent performs an action. Specifically, the congestion calculation is very expensive in terms of CPU time especially when it needs to be calculated very frequently. Additionally, the RL agent is trained in Python while GPlace is developed using C language. The interface between these two modules also introduces some runtime overhead to the whole flow.

Consequently, we propose to increase the number of Iteration per Action to reduce the runtime. Increasing IpA from 1 to higher numbers helps to reduce the runtime, as we need to re-calculate the state features every *n* iterations instead of every single iteration. To be more clear, with IpA more than 1, lines 5 - 13 of Algorithm 3 will be performed $|IpA|$ times, and the rest of the algorithm is performed only once. As a result, the total number of iterations (solved bipartite graph) will increase, while the main loop (line 1 - 24) is executed fewer times. Figure 5.6 shows that runtime is improved tremendously with increasing the number of IpA. Moreover, Table 5.9 presents the average runtime of the RL ISM with different IpAs over the 12 benchmarks. This table shows with increasing IpA to higher numbers like 40 and 50, the runtime improvement is decreased. With higher IpA, the objectives will improve at a slower rate and as a result, the terminating condition will be met later in some of the benchmarks (Algorithm 3, line 1).

IpA	1	10	20	30	40	50
Runtime (s)	50321	6512	4696	4256	3830	3665

Table 5.9: Effect of IpA on Runtime in DQN Model.

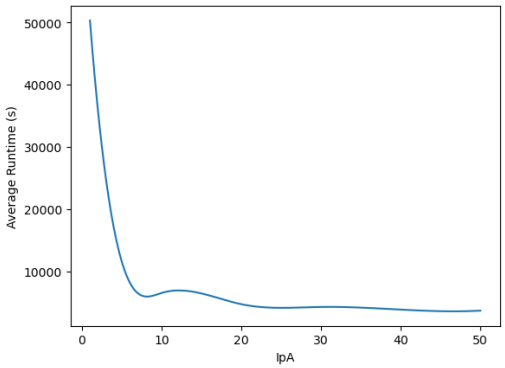


Figure 5.6: Effect of IpA on Runtime

5.3.2 Effect of Objective Weights:

In this experiment, the objective is to determine how the model's behavior is influenced by the objective weights λ. λ defines that the RL agent should put more effort towards which objective. When λ is set to 0, it means that our sole objective is external pins while λ equal to 1 means that we are only optimizing for WL.

We carried out this experiment with different numbers of λ while other parameters are fixed. Table 5.6 shows other parameters along with their associated values except for the IpA which is set to 30. Besides, we used ϵ-greedy strategy for action selection. Here again, we used 12 ISPD benchmarks to evaluate this experiment.

Table 5.10 shows the model's performance with different λs. When λ is increased from 0 to 1, wirelength is constantly improving which is desirable since we are putting more emphasis on this objective. However, with decreasing λ from 1 to 0, external pins first increase but as we continue decreasing the λ, external pins also start to deteriorate. This is because with increasing the weight of external pins in the reward function, the agent will choose wirelength actions less frequently while these moves can help to im-

prove external pins too. In other words, the agent has not learned about the future reward of each individual action in different stages of the episode. This is due to the fact that with the current ϵ-greedy strategy, the agent only explores different actions at the early stages of the episode, and afterward, it will follow a deterministic policy. So, it has never seen the benefits of selecting WL moves. Therefore, an RL model that explores the environment throughout the whole episode is needed. Figure 5.7 shows how the total reward changes with increasing λ. It shows that putting more emphasis on wirelength can lead to better total reward since wirelength moves are also helpful for optimizing external pins.

λ %	0	10	15	30	50	65	80	90	100
WL %	-9.8	1.8	3.2	5.4	6.4	6.8	7	7.3	7.8
EP %	20	20.1	20.8	23.1	23.8	23.6	22.4	21.9	18.4
Total	10.2	21.9	24	28.5	30.2	30.4	29.4	29.2	26.2

Table 5.10: Effect of λ in DQN Model.

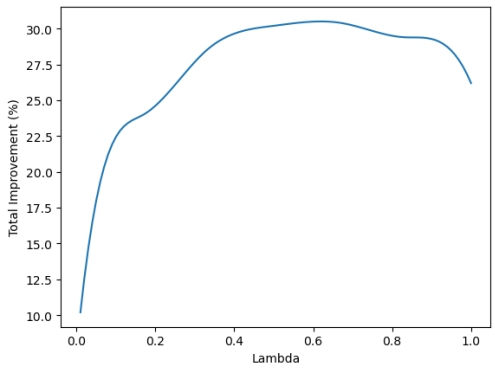


Figure 5.7: Effect of λ on Total Reward

5.3.3 Softmax Action Selection:

In this experiment, we want to investigate how the action selection strategy affects the model's performance. Table 5.6 shows the set of parameters used in this experiment except for the IpA which is set to 30. The other difference here is that we use a softmax strategy instead of ϵ-greedy. The objective is to determine if the softmax strategy can help the agent to explore the environment in different states during the whole episode.

Table 5.11 compares the amount of improvement in wirelength and external pins using different action selection strategies. This table shows that using the softmax strategy, external pins have improved by 1.3% while wirelength has deteriorated by 2%. Selected actions using the softmax strategy are depicted in Figure 5.8. This figure shows using the softmax strategy, all the actions have more chance to be selected in different states which helps the agent to explore the environment more effectively. On the other hand, a softmax strategy may fail to exploit what it has learned as this strategy is very dependent on the Q-values.

Figure 5.8 shows that actions that optimize external pins are selected more frequently compared to the ϵ-greedy strategy. This can justify why this strategy obtains better improvement in terms of external pins. On the other hand, wirelength moves are selected less frequently which can explain why the wirelength got worse using this strategy.

RL ISM	Epsilon-Greedy	Softmax
WL %	6.4	4.4
EP %	23.8	25.1
Total %	30.2	29.5

Table 5.11: Epsilon-Greedy vs Softmax

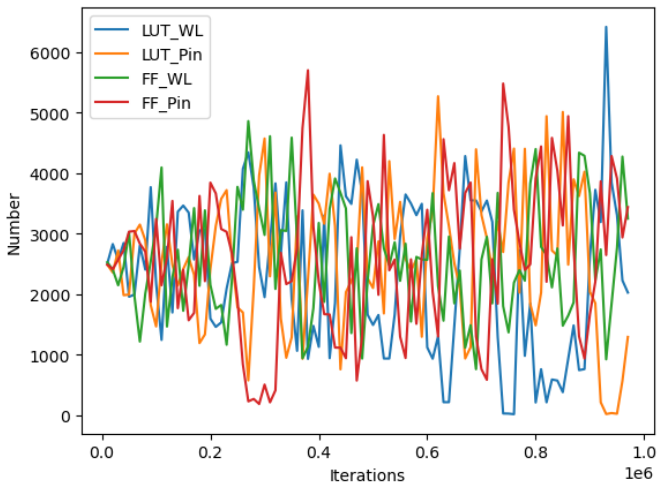


Figure 5.8: DQL Actions with Softmax Strategy

5.3.4 Model Evaluation

The Deep Q-Learning model was able to overcome the limitations of the Tabular Q-Learning where it can interact with an environment with continuous state-space. Moreover, using a replay buffer helps us to reuse past experiences which will help the DQL model to have a better sample efficiency. On the other hand, establishing a good balance between exploration and exploitation is a major problem of the DQL model. Besides,

this model is only able to learn deterministic policies which can lead to a sub-optimal solution in this problem. Moreover, the DQL model introduces more parameters to be configured. Tuning these parameters is a very challenging task as they can highly affect the training stability and QoR.

5.4 Advantage Actor-Critic (A2C) Model Results

In this section, we present the results of the Advantage Actor-critic model. First, we describe the initial result of this model. Then, we perform different experiments to explore how each one of them affects the model's performance. Table 5.12 shows the model's parameters and their associated values for the initial results presented in this section. Besides, we are using the reward equation 4.1 to generate our rewards and the agent is trained throughout a single episode for each benchmark.

Parameter	Value
Actor Learning Rate	0.01
Critic Learning Rate	0.01
Discount Factor	0.95
Iterations per Action	30
λ (Objective Weights)	0.5

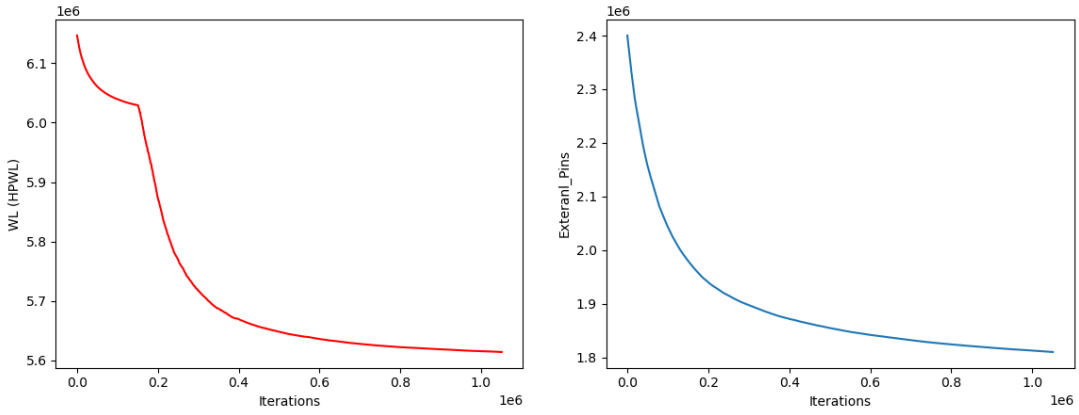
Table 5.12: Actor-Critic Model Parameters.

Table 5.13 presents the average improvement of the objectives over the 12 ISPD benchmarks. The A2C model was able to improve the wirelength by 6.07% and exter-

nal pins by 24.61%. Figure 5.9 shows how the objectives are changing throughout the episode, optimizing FPGA-10. The actions selected by the model in this episode are illustrated in Figure 5.10. This figure shows that at the very beginning of the episode, the probability associated with external pin moves is higher than the wirelength moves which leads to a slow improvement in wirelength. However, around iteration 100K, the wirelength moves selection frequency start to increase which accelerates the wirelength improvement rate. This can be observed in Figure 5.9a. External pins, Figure 5.9b, have a smoother improvement rate since wirelength moves can also help to optimize external pins.

Objectives	WL%	EP%	Total
Average Improvement	6.07	24.61	30.68

Table 5.13: Actor-Critic Model Performance.



(a) Wirelength (b) External Pins

Figure 5.9: Objectives Improvement During Episode

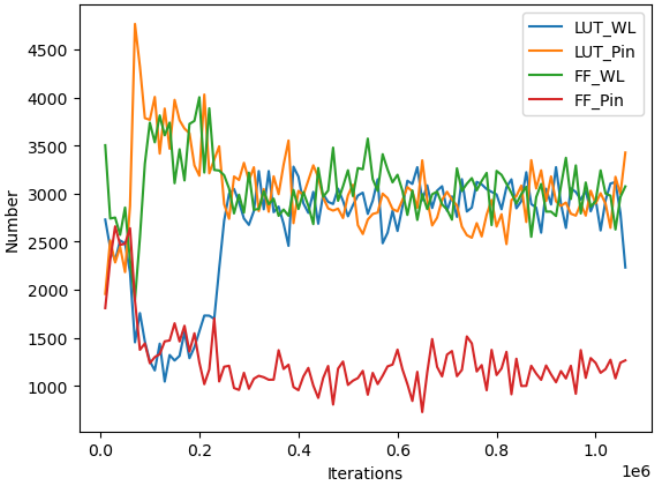


Figure 5.10: Actor-Critic Model Actions

The A2C model produces a probability distribution over the actions in different stages of the placement. Using this model, we can make sure that all of the actions are explored in different stages of the episode. This was one of the main reasons for migrating from value-based models to an actor-critic model.

5.4.1 Extending the Learning Process:

In this experiment, we propose to train the agent over the course of several episodes. The objective here is to see if the agent can utilize what it has learned in the previous episodes to achieve higher improvements. In this experiment, we reduced the learning rates of the actor and critic to 0.001. This is because now we can train the agent throughout several episodes so, we can provide the agent with more transition samples and we do not worry if the agent is not able to learn fast enough in a single episode.

In the previous experiments, all the weights in the value network and policy network

are discarded after finishing an episode, and the next episode starts with new random weights. However, we propose to save the weights that we obtained from placing one benchmark and use them as a starting point in the next episode (same benchmark). Our objective here is to see if the agent can make more smart decisions when it starts a new episode. Saving the model includes saving the Actor and Critic networks as well as the model's parameters like loss functions and the model's optimizer.

Figure 5.11 shows the training history of the Actor-Critic model optimizing FPGA-10. The blue line shows the total reward obtained in each episode and the orange line is the smoothed reward. In the training history, we cannot see a steady upward trend in the reward over 50 episodes where there is a high variance in total reward. However, when we look at fewer consecutive episodes, the agent is able to increase its total reward but then we see a sudden drop in the total reward. This is due to a phenomenon called catastrophic forgetting in which the agent forgets past states and needs to relearn patterns again. This is because the actor-critic algorithms are on-policy methods in which the agent is trained after each step with the samples produced at that step and all the previous samples are discarded. These samples are highly correlated which adversely affects the training process. Another reason is that we are using function approximation (Critic) for the target values. Using more robust target values similar to the target network used in the DQL model can help to stabilize the learning process but then again there will be a new parameter to be tuned.

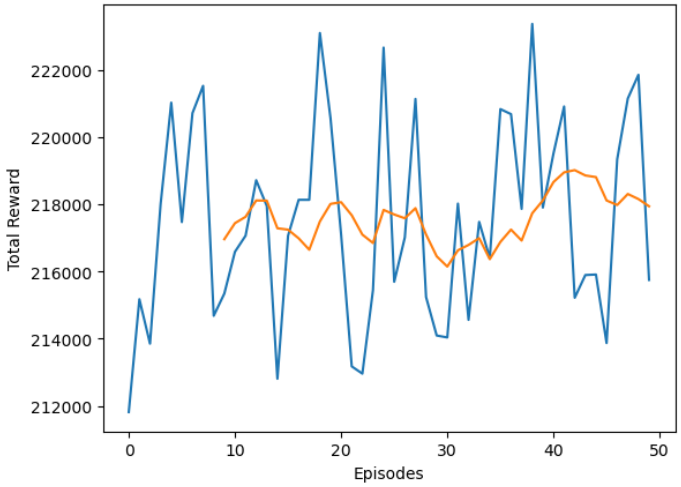


Figure 5.11: A2C Training History

5.4.2 Alternative Reward Function

In this experiment, our objective is to see how the model's behavior is influenced by the reward function. The reward function is a very important part of the RL methods in which we define our objectives. A clear reward signal can help the agent to reach the optimal solution faster while a vague reward signal can stop the agent from learning anything meaningful.

Here, we keep all the parameters the same as Table 5.12, except for the learning rate which is set to 0.001. We also propose to use the reward equation 4.2 to produce the reward signal. This reward function penalizes the agent if the improvement in either of the objectives drops below a specific threshold. With the previous reward function, equation 4.1, the agent receives either a positive reward or zero rewards. Our objective was here to create a clearer difference between rewards in different states during the episode to help the agent distinguish better between these states and avoid the states that

can potentially reduce the total reward.

Figure 5.12 shows the training history of the A2C model optimizing FPGA 10. The blue line shows the total reward obtained in each episode and the orange line is the smoothed reward. Here, There is still a high variance in the total reward which is due to the reasons explained in the previous experiment.

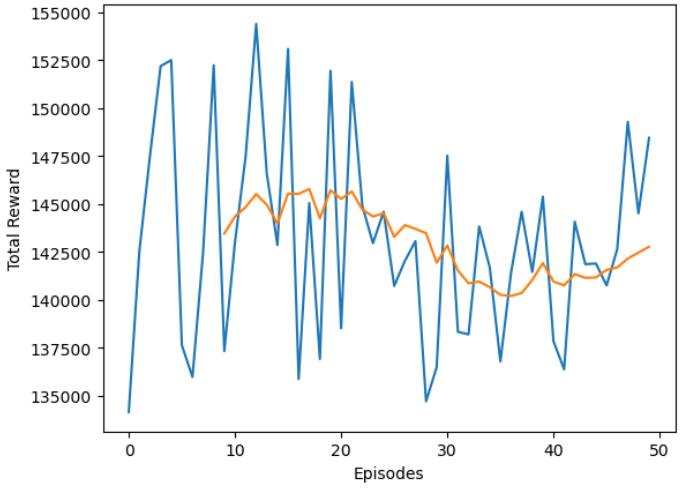


Figure 5.12: A2C Training History with Reward Equation 4.2

Table 5.14 presents the average improvement in wirelength, 6.03%, and external pins, 21.45%, over 12 benchmarks in which each of the benchmarks is trained 50 episodes. This model was able to improve the wirelength very similar to the initial result of the A2C model. However, it could not improve the external pins that well. This happens because the external pin is usually improved at a higher rate compared to the wirelength. Therefore, the reward associated with external pin moves becomes negative faster, and as a result, the agent is less likely to choose these actions in later stages of the episode. A more general problem with the proposed RL framework is our action space. The actions

that we have in the detailed placement stage, change the placement very slowly and their outcome can be similar in terms of state transition. Therefore the agent will get into trouble learning how each action takes us from one state to another which can hurt the learning process.

Objectives	WL%	EP%	Total
Average Improvement	6.03	23.45	29.48

Table 5.14: Actor-Critic Model Performance with Reward Equation 4.2

Figure 5.13 shows the distribution of the action in episode 50. The agent starts by exploring all the actions. However, as it goes through the episode, the probability associated with external pin moves is decreased and these actions are less likely to be selected. This can explain why the external pins improvement is worse than the result introduced in Table 5.13.

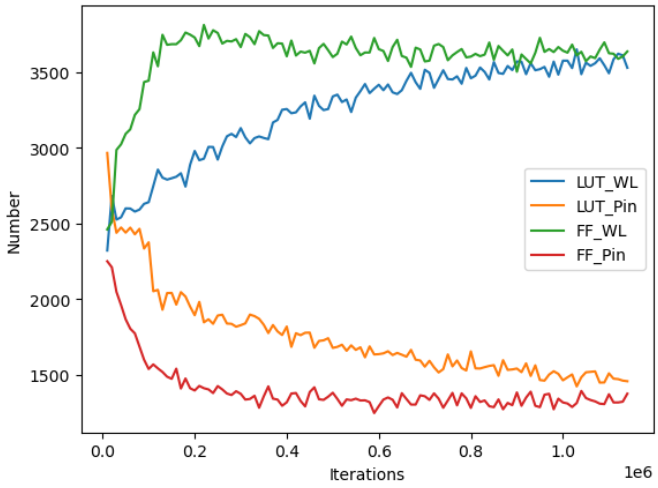


Figure 5.13: Action Distribution with in Episode 50

5.4.3 Model Evaluation

The Advantage Actor-Critic model was capable of exploring the environment through the whole episode by producing a probability distribution over the action space. Besides, this model was able to establish a balance between optimizing wirelength and external pins by following a stochastic policy rather than a deterministic policy.

However, this model suffers from catastrophic forgetting. In this model, we train the agent using the most recent samples. Therefore, as we go through the episode, the agent gradually forgets what it has learned at the early stages leading to an unstable learning process. Additionally, another problem with the proposed RL framework in this thesis is that the outcome of each action is not very clear in terms of the state transition since each individual action changes the placement solution very slowly. As a result, the agent can capture which actions are more rewarded, but it gets into trouble learning how each action will take us from one state to another.

5.5 Comparing GPlace3.0 ISM vs RL ISM Models

In this section, we are going to compare our RL models to each other and GPlace3.0 ISM in terms of QoR, and Runtime. Here we compare the results of each model with its best configuration. Table 5.15 presents the parameters associated with each model. To have a fair comparison, we use the result of the A2C model which is trained over a single benchmark like the other models. Moreover, to be able to compare different models in terms of runtime, we report the runtime of each model without calculating Mean Wire Congestion as a state feature. Eliminating congestion does not affect the QoR. In addition to the RL ISM models and GPlace3.0 ISM, we compare our models

with a Random ISM model too. The Random ISM model has the exact same flow as the RL ISM models except that the RL agent is bypassed and actions are selected with equal probabilities. The Random ISM model is a better baseline compared to the GPlace3.0 ISM because the GPlace3.0 ISM has a quite different flow from RL models.

Below, we first compare different models in terms of wirelength improvement, section 5.5.1. Then, the External Pins improvements are compared in section 5.5.2. In section 5.5.3, the runtime of different models is presented and discussed. Finally, we evaluate the total performance of each model in section 5.5.4.

Parameter	TQL	DQN	A2C	Random ISM
Learning Rate	0.01	0.01	0.01	N.A
Discount Factor	0.5	0.95	0.95	N.A
Iterations per Action	20	30	30	20
λ (Objective Weights)	0.5	0.5	0.5	N.A

Table 5.15: Model's Parameters for Comparison

5.5.1 Wirelength Comparison

Figure 5.14 presents the wirelength improvement of different models throughout a single episode. This figure shows that the DQL model outperformed the other RL models in wirelength improvement. The A2C model has the next best performance and the TQL model obtained the least amount of improvement among the RL models. Moreover, all the RL models significantly outperformed the Random ISM. The GPlace3.0 ISM has still the best performance among the models presented in this section.

Table 5.16 presents the amount of wirelength improvement for all 12 ISPD benchmarks using different models. On average, the DQL, A2C, and TQL models improved the wirelength by 6.4%, 6.07%, and 3.8%, respectively. Our results show that these RL models model outperformed the Random ISM by 284%, 270%, and 168%. Besides, these models could achieve QoR comparable to the GPlace3.0 ISM. The GPlace3.0 ISM still outperforms the DQL, A2C, and TQL models by 3%, 8.7%, and 173%, respectively.

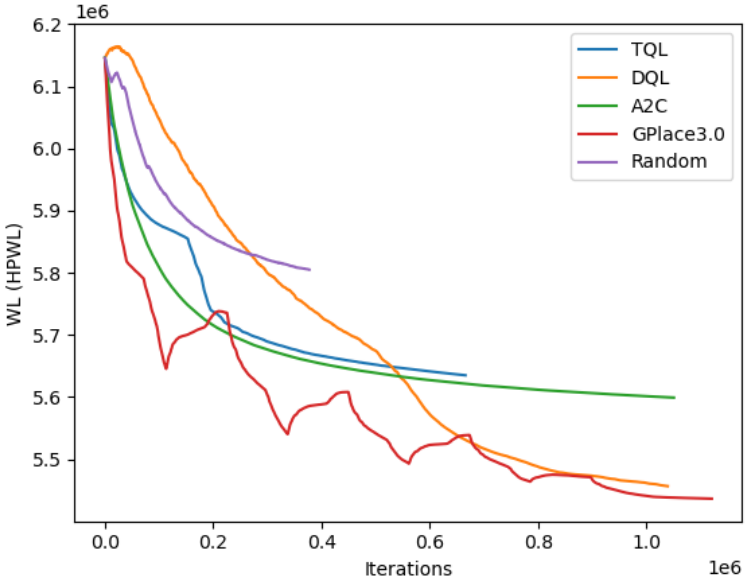


Figure 5.14: WL of each model during placement

Benchmark	GPlace3.0	Random	TQL	DQN	A2C
FPGA-1	15.46	6.54	3.55	13.10	13.50
FPGA-2	8.84	4.84	5.20	7.82	7.7
FPGA-3	5.44	2.2	3.23	4.77	4.3
FPGA-4	5.85	0.66	3.07	4.51	4.2
FPGA-5	5.28	0.40	2.03	3.24	3.29
FPGA-6	9.35	-0.14	3.2	6.48	6.68
FPGA-7	6.84	0.38	3.53	5.3	5.02
FPGA-8	4.14	0.73	2.17	3.55	3.32
FPGA-9	5.11	0.37	2.59	4.38	3.94
FPGA-10	11.67	5.28	8.31	11.35	8.97
FPGA-11	5.33	1.38	2.68	4.11	4.02
FPGA-12	9.62	4.45	6.17	8.36	7.96
Average	6.60	2.25	3.8	6.40	6.07

Table 5.16: Wirelength Improvement (%) in Different Models

5.5.2 External Pins Comparison

Figure 5.15 presents the external Pins improvement of different models throughout a single episode. This figure shows that the TQL model outperformed the other RL models in external pins improvement. The A2C model has the next best performance and the DQL model obtained the least amount of improvement among the RL models. Besides, all the RL models significantly outperformed the Random ISM. The TQL model even outperforms the GPlace3.0 ISM, but the GPlace3.0 ISM outperforms the A2C and DQL models.

Table 5.17 presents the amount of external pins improvement for all 12 ISPD benchmarks using different models. On average, The TQL, A2C, and TQL models improved the external pins by 26.35%, 24.61&, and 23.8%, respectively. Our results show that these RL models model outperformed the Random ISM by 207%, 193%, and 187%. Besides, these models could achieve QoR comparable or even better than the GPlace3.0 ISM. The DQL model outperformed the GPlace3.0 ISM by 5.9% while the GPlace3.0 ISM outperforms the A2C and DQL models by 1%, and 4.5%, respectively.

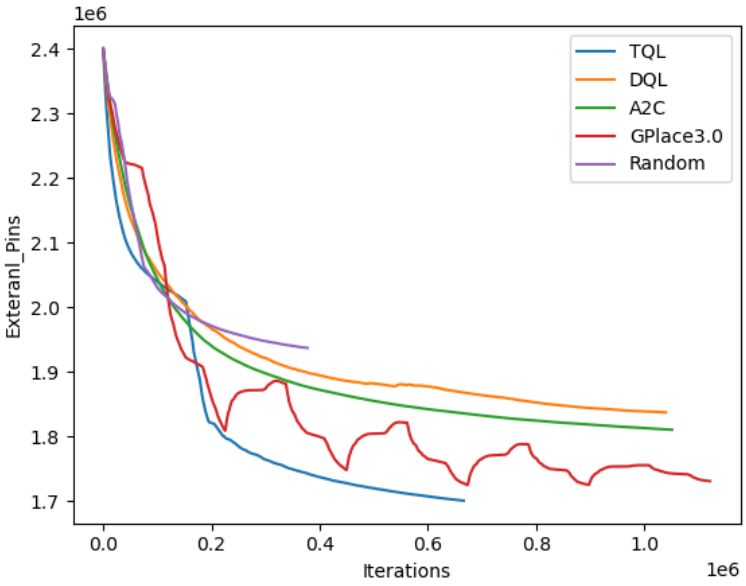


Figure 5.15: EPs of each model during placement

Benchmark	GPlace3.0	Random	TQL	DQN	A2C
FPGA-1	26.84	20.14	17.83	25.92	26.20
FPGA-2	24.49	16.00	26.36	22.15	23.69
FPGA-3	22.26	10.18	24.86	21.48	23.09
FPGA-4	24.73	8.90	27.5	24.66	25.47
FPGA-5	24.52	7.61	26.75	25.05	26.36
FPGA-6	23.05	10.87	26.38	21.82	22.66
FPGA-7	24.35	9.23	26.83	25.45	24.9
FPGA-8	25.38	7.41	27.99	24.99	24.14
FPGA-9	22.00	9.64	25.1	21.65	21.39
FPGA-10	27.89	18.54	29.16	21.64	24.16
FPGA-11	23.74	10.20	26.13	22.85	24.92
FPGA-12	29.17	23.53	31.27	23.75	28.33
Average	24.88	12.7	26.35	23.8	24.61

Table 5.17: External Pins Improvement (%) in Different Models

5.5.3 Runtime Comparison

Table 5.18 presents the runtime of different models for all 12 ISPD benchmarks. On average, the runtime associated with the TQL, DQL, and A2C models is 188s, 270s, and 325s, respectively. Our results show that all the RL models are slower than the Random ISM by 37%, 97%, and 237%. However, the RL models significantly outperformed the GPlace3.0 ISM in terms of runtime. The TQL, DQL, and A2C models improved the runtime by 71%, 58%, and 50% compared to the GPlace3.0 ISM.

Benchmark	GPlace3.0	Random	TQL	DQN	A2C
FPGA-1	50.34	21.62	28.12	33.34	39.57
FPGA-2	116.97	36.25	65.37	75.53	84.76
FPGA-3	349.47	66.72	96.46	190.19	203.08
FPGA-4	359.02	63.48	78.28	162.84	225.01
FPGA-5	396.44	145.18	125.19	204.30	243.47
FPGA-6	686.94	93.66	165.53	279.91	295.48
FPGA-7	741.45	135.35	248.71	317.15	394.05
FPGA-8	796.46	184.57	281.41	388.32	386.19
FPGA-9	940.06	240.84	340.25	383.18	444.60
FPGA-10	1114.36	195.36	230.76	357.22	458.51
FPGA-11	937.47	200.01	256.45	370.22	737.75
FPGA-12	1313.49	271.43	345.30	489.11	387.39
Average	650	137	188	270	325

Table 5.18: Runtime (s) in Different Models

5.5.4 Overall Comparison

Table 5.18 presents the average total improvement of different models as well as the average runtime. The average total improvement is the sum of the average wirelength improvement and the average external pins improvement. The A2C, DQL, and TQL models obtained the average improvement of 30.68%, 30.2&, and 30.15%, respectively. Our results show that these RL models model outperformed the Random ISM by 205%, 202%, and 201%. On the Other hand, the GPlace3.0 ISM outperforms the A2C, DQL, and TQL models by 2.6%, 4.2%, 4.4%. Among the RL models, the A2C model outperforms the DQL, and TQL models by 1.5%, and 1.7% respectively. The A2C model, which has the highest average improvement, reduced the runtime by 50% compared to the GPlace3.0 ISM while the QoR is deteriorated by 2.6%.

Models	GPlace3.0	Random	TQL	DQN	A2C
Average Improvement (%)	31.48	14.95	30.15	30.2	30.68
Average Runtime (s)	1887	137	188	270	325

Table 5.19: Average performace of all Models.

Chapter 6

Conclusions and Future Directions

This chapter concludes this thesis in section 6.1, underlying the achievements of our proposed framework. In section 6.2, potential directions for future works are proposed.

6.1 Conclusion

In this thesis, a Reinforcement Learning technique was proposed to further enhance the detailed placement stage of the GPlace3.0 FPGA placement flow. An RL framework was integrated into the detailed placement stage of GPlace3.0 to dynamically select the optimization strategies with respect to the different characteristics of each circuit placement. Several RL models were investigated and the advantages and drawbacks of each model were highlighted. These RL techniques are able to explore and exploit the solution space compared to the traditional heuristic approaches leading to a significant reduction in runtime (up to 50%) while preserving the solution quality.

6.2 Future Works

There are several aspects of the traditional CAD flow that suffer from the static flow and Reinforcement Learning can be employed to add flexibility and smartness to the decision-making process. Here we propose some directions for future works to further improve FPGA CAD flow.

- Using RL for Global Analytical Placement: There are several wire models including Star+, B2B, and Quadratic. The RL can be used to determine which wire model to choose for each benchmark during the global placement to obtain a better solution.

- Congestion management is another stage of CAD flow that can employ an RL framework. Mainly, RL can be used to decide whether to increase or not the size of logic cells during the Cell Inflation to resolve congestion, thus enabling congestion management to be treated as a dynamic optimization problem.

- The legalization phase is yet another important step of analytical placement that can be a good candidate for incorporating RL algorithms to enhance solution quality. The RL algorithm employed can self-learn the best strategy to legalize modules. For example, the RL algorithm can decide when and where to apply exact legalization versus rough legalization.

Appendix A

Glossary

CAD	: Computer Aided Design
CMOS	: Complementary Metal Oxide Semiconductor
DA	: Design Automation
FPGA	: Field Programmable Gate Array
HPWL	: Half Perimeter Wire Length
ILP	: Integer Linear Programming
IP	: Intellectual Property
MCNC	: Microelectronics Center of North Carolina
NP-hard	: Non Deterministic Polynomial Hard
RTL	: Register Transfer Logic
SoC	: System on Chip
VHDL	: Very High Speed Integrated Circuit Hardware Description Language
VLSI	: Very Large Scale Integration

Bibliography

[1] Z. Abuowaimer, D. Maarouf, T. Martin, J. Foxcroft, G. Grwal, S. Areibi, and A. Vannelli, "Gplace3.0: Routability-driven analytic placer for ultrascale fpga architectures," *ACM Transactions on Design Automation of Electronic Systems*, vol. 23, pp. 1–33, 10 2018.

[2] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*. Cambridge, MA, USA: A Bradford Book, 2018.

[3] A. Boutros and V. Betz, "Fpga architecture: Principles and progression," *IEEE Circuits and Systems Magazine*, vol. 21, no. 2, pp. 4–29, 2021.

[4] M. A. Elgammal, K. E. Murray, and V. Betz, "Learn to place: Fpga placement using reinforcement learning and directed moves," *2020 International Conference on Field-Programmable Technology (ICFPT)*, pp. 85–93, 2020.

[5] N. Kapre, H. Ng, K. Teo, and J. Naude, "Intime: A machine learning approach for efficient selection of fpga cad tool parameters," ser. FPGA '15. New York, NY, USA: Association for Computing Machinery, 2015, p. 2326. [Online]. Available: https://doi.org/10.1145/2684746.2689081

[6] K. E. Murray, S. Whitty, S. Liu, J. Luu, and V. Betz, "Timing-driven titan: Enabling large benchmarks and exploring the gap between academic and commercial cad," *ACM Trans. Reconfigurable Technol. Syst.*, vol. 8, no. 2, mar 2015. [Online]. Available: https://doi.org/10.1145/2629579

[7] A. Khatkhate, C. Li, A. R. Agnihotri, M. C. Yildiz, S. Ono, C.-K. Koh, and P. H. Madden, "Recursive bisection based mixed block placement," in *Proceedings of the 2004 International Symposium on Physical Design*, ser. ISPD '04. New York, NY, USA: Association for Computing Machinery, 2004, p. 8489. [Online]. Available: https://doi.org/10.1145/981066.981084

[8] K. E. Murray, O. Petelin, S. Zhong, J. M. Wang, M. Eldafrawy, J.-P. Legault, E. Sha, A. G. Graham, J. Wu, M. J. P. Walker, H. Zeng, P. Patros, J. Luu, K. B. Kent, and V. Betz, "Vtr 8: High-performance cad and customizable fpga architecture modelling," *ACM Trans. Reconfigurable Technol. Syst.*, vol. 13, no. 2, may 2020. [Online]. Available: https://doi.org/10.1145/3388617

[9] W. Li, Y. Lin, and D. Z. Pan, "elfplace: Electrostatics-based placement for large-scale heterogeneous fpgas," in *2019 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2019, pp. 1–8.

[10] T. Martin, D. Maarouf, Z. Abuowaimer, A. Alhyari, G. Grewal, and S. Areibi, "A flat timing-driven placement flow for modern fpgas," in *2019 56th ACM/IEEE Design Automation Conference (DAC)*, 2019, pp. 1–6.

[11] G. Chen, C. Pui, W. Chow, K. Lam, J. Kuang, E. Young, and B. Yu, "RippleFPGA: Routability-Driven Simultaneous Packing and Placement for Modern FP-

GAs," *IEEE Transactions on CAD and Systems*, vol. 37, no. 10, pp. 2022–2035, October 2018.

[12] A. Mametjanov, P. Balaprakash, C. Choudary, P. D. Hovland, S. M. Wild, and G. Sabin, "Autotuning fpga design parameters for performance and power," in *2015 IEEE 23rd Annual International Symposium on Field-Programmable Custom Computing Machines*, 2015, pp. 84–91.

[13] Q. Liu, J. Ma, and Q. Zhang, "Neural network based pre-placement wirelength estimation," in *2012 International Conference on Field-Programmable Technology*, 2012, pp. 16–22.

[14] J. Kober, J. A. Bagnell, and J. Peters, "Reinforcement learning in robotics: A survey," *The International Journal of Robotics Research*, vol. 32, no. 11, pp. 1238–1274, 2013.

[15] D. Silver, T. Hubert, J. Schrittwieser, I. Antonoglou, M. Lai, A. Guez, M. Lanctot, L. Sifre, D. Kumaran, T. Graepel, T. Lillicrap, K. Simonyan, and D. Hassabis, "A general reinforcement learning algorithm that masters chess, shogi, and go through self-play," *Science*, vol. 362, no. 6419, pp. 1140–1144, 2018. [Online]. Available: https://www.science.org/doi/abs/10.1126/science.aar6404

[16] K. Murray and V. Betz, "Adaptive fpga placement optimization via reinforcement learning," 09 2019, pp. 1–6.

[17] U. Farooq, N. Ul Hasan, I. Baig, and M. Zghaibeh, "Efficient fpga routing using reinforcement learning," 05 2021, pp. 106–111.

[18] Xilinx, "ISPD 2016 Routability-Driven FPGA Placement Contest," http://www.ispd.cc/contests/16/ispd2016_contest.html, 2016.

[19] P. Spindler, U. Schlichtmann, and F. M. Johannes, "Kraftwerk2a fast force-directed quadratic placement approach using an accurate net model," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 27, no. 8, pp. 1398–1411, 2008.

[20] Xilinx, "UltraScale Architecture Configurable Logic Block User Guide," 2017.

[21] M. Xu, G. Grewal, and S. Areibi, "Starplace: A new analytic method for fpga placement," *Integration*, vol. 44, no. 3, pp. 192–204, 2011. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167926011000253

[22] W. Li, S. Dhar, and D. Z. Pan, "Utplacef: A routability-driven fpga placer with physical and congestion aware packing," in *2016 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2016, pp. 1–7.

[23] O. Vinyals, I. Babuschkin, W. Czarnecki, M. Mathieu, A. Dudzik, J. Chung, D. Choi, R. Powell, T. Ewalds, P. Georgiev, J. Oh, D. Horgan, M. Kroiss, I. Danihelka, A. Huang, L. Sifre, T. Cai, J. Agapiou, M. Jaderberg, and D. Silver, "Grandmaster level in starcraft ii using multi-agent reinforcement learning," *Nature*, vol. 575, 11 2019.

[24] L. McMurchie and C. Ebeling, "Pathfinder: A negotiation-based performance-driven router for fpgas," in *Third International ACM Symposium on Field-Programmable Gate Arrays*, 1995, pp. 111–117.

[25] A. Alhyari, S. Areibi, Z. Abuowaimer, G. Grewal, and A. Elshamli, "A deep learning framework to predict routability for fpga circuit placement," 09 2019.

[26] D. Maarouff, A. Shamli, T. Martin, G. Grewal, and S. Areibi, "A deep-learning framework for predicting congestion during fpga placement," in *2020 30th International Conference on Field-Programmable Logic and Applications (FPL)*, 2020, pp. 138–144.

[27] K. E. Murray and O. Petelin, "Vtr 8: High performance cad and customizable fpga architecture modelling," 2020.

[28] Y.-H. Huang, Z. Xie, G.-Q. Fang, T.-C. Yu, H. Ren, S.-Y. Fang, Y. Chen, and J. Hu, "Routability-driven macro placement with embedded cnn-based prediction model," in *2019 Design, Automation Test in Europe Conference Exhibition (DATE)*, 2019, pp. 180–185.

[29] Q. Zhou, X. Wang, Z. Qi, Z. Chen, Q. Zhou, and Y. Cai, "An accurate detailed routing routability prediction model in placement," in *2015 6th Asia Symposium on Quality Electronic Design (ASQED)*, 2015, pp. 119–122.

[30] Z. Qi, Y. Cai, and Q. Zhou, "Accurate prediction of detailed routing congestion using supervised data learning," in *2014 IEEE 32nd International Conference on Computer Design (ICCD)*, 2014, pp. 97–103.

[31] R. Kirby, K. Nottingham, R. Roy, S. Godil, and B. Catanzaro, "Guiding global placement with reinforcement learning," 2021.

[32] A. Mirhoseini, A. Goldie, M. Yazgan, J. Jiang, E. Songhori, S. Wang, Y.-J. Lee,

E. Johnson, O. Pathak, S. Bae, A. Nazi, J. Pak, A. Tong, K. Srinivasa, W. Hang, E. Tuncer, A. Babu, Q. V. Le, J. Laudon, R. Ho, R. Carpenter, and J. Dean, "Chip placement with deep reinforcement learning," 2020.

[33] T. Qu, Y. Lin, Z. Lu, Y. Su, and Y. Wei, "Asynchronous reinforcement learning framework for net order exploration in detailed routing," in *2021 Design, Automation Test in Europe Conference Exhibition (DATE)*, 2021, pp. 1815–1820.

[34] H. Liao, W. Zhang, X. Dong, B. Poczos, K. Shimada, and L. B. Kara, "A deep reinforcement learning approach for global routing," 2019.

[35] T. Qu, Y. Lin, Z. Lu, Y. Su, and Y. Wei, "Asynchronous reinforcement learning framework for net order exploration in detailed routing," in *2021 Design, Automation Test in Europe Conference Exhibition (DATE)*, 2021, pp. 1815–1820.

[36] H. Li, G. Chen, B. Jiang, J. Chen, and E. F. Y. Young, "Dr. cu 2.0: A scalable detailed routing framework with correct-by-construction design rule satisfaction," in *2019 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2019, pp. 1–7.

[37] H. Ren and M. Fojtik, "Standard cell routing with reinforcement learning and genetic algorithm in advanced technology nodes," in *2021 26th Asia and South Pacific Design Automation Conference (ASP-DAC)*, 2021, pp. 684–689.

[38] F. WANG and W. ZHENG, "To rip or not to rip: A reinforcement learning-based rip-up and reroute algorithm for global routing," 2020. [Online]. Available: https://openreview.net/forum?id=jjdngaZiwVb

[39] R. Kirby, K. Nottingham, R. Roy, S. Godil, and B. Catanzaro, "Guiding global placement with reinforcement learning," 2021.

[40] H. Szentimrey, A. Al-Hyari, J. Foxcroft, T. Martin, D. Noel, G. Grewal, and S. Areibi, "Machine learning for congestion management and routability prediction within fpga placement," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 25, no. 5, aug 2020. [Online]. Available: https://doi.org/10.1145/3373269

[41] V. Mnih, A. P. Badia, M. Mirza, A. Graves, T. Harley, T. P. Lillicrap, D. Silver, and K. Kavukcuoglu, "Asynchronous methods for deep reinforcement learning," in *Proceedings of the 33rd International Conference on International Conference on Machine Learning - Volume 48*, ser. ICML'16. JMLR.org, 2016, p. 19281937.

[42] F. Chollet *et al.* (2015) Keras. [Online]. Available: https://github.com/fchollet/keras