technische universität dortmund

Arbeitsgebiet Datentechnik Information Processing Lab

AI-Based SI/PI/EMC-Compliant PCB Design

Implementation of ML/AI Modules to Support Interference-Resistant Design of Microelectronic Systems

Multi-Agent Reinforcement Learning (MARL) Method for Decoupling Capacitors (Decaps) Placement and Optimization of the PCB-Power Delivery Network (PDN)

Project Group WISE 2023/24

submitted to

Faculty of Electrical Engineering and Information Technology

AG Datentechnik / Information Processing Lab

Technische Universität Dortmund

by

Darshana Anil Hosurkar, Emmanuel Hernandez and ZubairAhmed Ansari

Date of Submission: June 16, 2024

Academic Supervisors:
Dr. Ing. Werner John
Prof. Dr. Ing. Jürgen Götze
M.Sc. Emre Ecik
M.Sc. Nima Ghafarian Shoaee
M.Sc. Julian Withöft

Abstract

This project presents an innovative approach for optimizing the placement of decoupling capacitors on printed circuit boards (PCBs) using machine learning integrated with multi-agent systems. The primary goal is to assist users in selecting optimal capacitor positions to enhance PCB performance. Our methodology employs the QMIX algorithm, a sophisticated reinforcement learning technique for multi-agent environments. We implemented this approach on PCBs with two configurations: one with 12 decoupling capacitors and another with 8, featuring both square and rectangular shapes. The QMIX algorithm efficiently coordinates the actions of multiple agents, each representing a capacitor, to identify the best positions that minimize the number of capacitors and maximize overall efficiency. Experimental results demonstrate the efficacy of our method. This work paves the way for advanced PCB design automation, potentially reducing design time and improving electronic circuit reliability.

Contents

Abbreviations

AI	Artificial Intelligence
CTDE	Centralized Training with Decentralized Execution
DECAPS	Decoupling Capacitors
DQN	Deep Q-Network
ICs	Integrated Circuits
MADDPG	Multi-Agent Deep Deterministic Policy Gradient
MARL	Multi-Agent Reinforcement Learning
PCBs	Printed Circuit Boards
PDN	Power Delivery Network
PI	Power Integrity
QMIX	Q-value Mixing
QPLEX	Q-value Propagation Learning EXtension
QTRAN	Q-value Transformed Reinforcement Learning
RL	Reinforcement Learning
RLlib	Reinforcement Learning Library
VDNs	Value Decomposition Networks

Introduction

Printed Circuit Boards (PCBs) are the backbone of the electronics industry, providing a foundation for efficient and reliable circuit design. A PCB is an electronic board with metal circuits embedded within it which connect different components on the device. It comprises of layers that are separately etched and laminated together to form traces that carry information from one part of the board to the other. Their compact layouts allow for the assembly of intricate electronic components, streamlining manufacturing processes and reducing manual wiring efforts. PCBs ensure optimal electrical connectivity, signal integrity and scalability across diverse applications, enhancing interchangeability, ease of repair and cost efficiency in electronic production [20].

This report focuses on a critical aspect of PCB design: the placement of decoupling capacitors (decaps) in the PCB-Power Delivery Network (PDN). The trend towards faster interfaces and lower power consumption has resulted in devices that are increasingly susceptible to disturbances from power and signal lines. This is where decaps come to play with a pivotal role in mitigating noise, stabilizing power supplies and enhancing overall circuit performance. Place strategically near power-hungry components, decaps absorb and release charge to counteract voltage fluctuations, ensuring a steady and noise-free power distribution [1].

The integration of Reinforcement Learning (RL) methods into PCB design processes offers a dynamic approach to optimizing decaps placement. RL algorithms can enhance automation, accelerate testing, and optimize routing strategies, thus contributing to more efficient designs and streamlined production workflows. This collaborative approach aims to improve the overall efficiency of the design process, leading to better space utilization, reduced signal interface, and enhanced overall performance. By exploring Multi-Agent Reinforcement Learning (MARL) methods in the specific context of decaps placement, where each agent represents an IC on the board, this report aims to achieve a coordinated strategy between the two ICs in order to maintain the impedance below threshold. This ensures the use of the least number of decaps while maintaining optimal performance of the PCB.

1 FUNDAMENTALS

In the rapidly evolving field of electronics and computing, ensuring efficient and reliable power distribution to various components on a device is crucial. A Power Delivery Network (Power Delivery Network (PDN)), also known as a power distribution network, plays a key role in this context [4]. The PDN is responsible for distributing power from the power supply to different parts of a device, primarily through printed circuit boards (PCBs) [19]. Historically, PDNs were designed with significant safety margins to prevent timing or functional failures. However, as devices have become more complex and operating frequencies have increased, these margins can no longer be afforded. This has led to a greater emphasis on precise PDN design and power integrity analysis to meet the stringent requirements of modern high-performance systems while minimizing costs and space.
Central to optimizing PDNs are decoupling capacitors (Decoupling Capacitors (DECAPS)), which stabilize the voltage supplied to integrated circuits (Integrated Circuits (ICs)) by filtering out noise and providing a steady power source. Proper placement and selection of these capacitors are critical to maintaining a consistent power supply and reducing impedance. Additionally, advanced techniques like Reinforcement Learning (Reinforcement Learning (RL)) and Multi-Agent Reinforcement Learning (Multi-Agent Reinforcement Learning (MARL)) have been explored to automate and optimize various aspects of PCB design, including the strategic placement of decaps [23]. These machine learning approaches enable the system to learn from experience and dynamically adjust the PDN for optimal performance, addressing the challenges posed by the increasing demands on power delivery in modern electronic devices.

1.1 Power Delivery Network and optimization Techniques

The Power Delivery Network (PDN) is pivotal in modern electronics, facilitating the distribution of power from a source to various components on a PCB. To ensure stability and reliability, decoupling capacitors (decaps) are strategically placed within the PDN to stabilize voltage and mitigate noise. Optimal decap placement is essential to meet target impedance requirements while minimizing cost and space utilization. Reinforcement Learning (RL) techniques have emerged as effective tools for automating PDN optimization processes, including power plane design and decap placement, based on target impedance at specific IC ports [9]. Multi-Agent Reinforcement Learning (MARL) extends RL to

scenarios where multiple agents collaborate to achieve common or individual goals. MARL presents opportunities for coordinated PDN optimization, albeit with challenges such as credit assignment and decentralized decision-making [10]. However, by addressing these challenges, MARL holds promise for revolutionizing PDN optimization by enabling agents to learn effective strategies for maximizing cumulative rewards while ensuring electronic design stability and performance.

1.1.1 Power Delivery Network (PDN)

A Power Delivery network (PDN) also know as a power distribution network, is a system that distributes power from a power supply to various components on a device. The PDN generally starts at power supply and is transferred by wires to a PCB. In the early times, the PDN was over-designed by adding significant margins to all aspects of the network. Thus, ensuring that timing or functional failures were not caused by an inadequate design. Recently, that level of margin can no longer be tolerated and much more attention is being paid to the PDN by performing thorough analysis sometimes termed as the power integrity analysis [7]. The goal of a PDN is to minimize the number of decaps while meeting the target impedance requirements. This is because each decap adds to the cost and takes up some space on the PCB [3]. With the increase in the operating frequencies and current loads in current high-performance computing systems, a robust PDN design which provides a stable power supply to integrated circuits (ICs) has become very challenging [10].

1.1.2 Decoupling Capacitors (DECAPS)

A decoupling capacitor is not a specific component but a term describing the function of a capacitor in an electronic circuit. It is used to stabilize the voltage on the power supply chain by either absorbing or releasing the charges when the voltage is beyond or below a certain threshold respectively [5]. Components like integrated circuits rely on the input voltage being as stable as possible. Thus, one can protect these sensitive chips by filtering out any excess noise and creating a steady source of power with the help of decaps [2]. Lately, decaps have been used extensively in order to optimize PDNs [9].

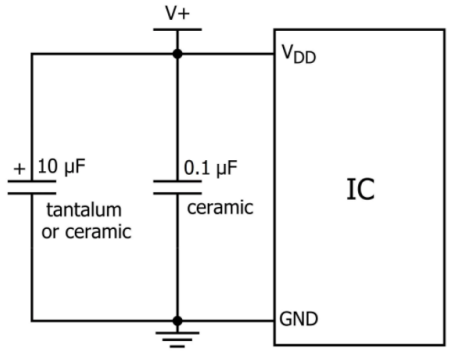


Figure 1.1: An Application Representing an IC and Decaps [1]

[1]Source: [2]

Figure 1.1 illustrates a common application, where a 10μF capacitor positions farther away from the IC addresses low-frequency changes in the input voltage, providing a smoothing effect. Conversely, a 0.1μF capacitor, placed closer to the IC, focuses on mitigating high-frequency noise. The combined action results in a consistent and smooth voltage supply to the IC [2].

1.1.3 Capacitor Placement

Capacitor placement on a PCB significantly impacts a PDN. Capacitors play a crucial role in stabilizing voltage, reducing noise and ensuring a consistent power supply to sensitive components. Our project focuses on identifying optimal capacitor placements on PCBs.

Type	Parameters		
	Capacitance [nF]	ESL [pH]	ESR [$\mu\Omega$]
1	100	222	8.9
2	47	154	21.4

Table 1.1: Decaps Library

Table 1.1 depicts the set of decaps alongside its parameters used for the implementation in the 8 and 12 port board design. These decaps have been used for both, testing the environment and for the algorithm. One primary reason for choosing these types is that large capacitors tend to have a larger internal impedance at high frequencies, longer lead structures with higher series inductance.

1.1.4 Reinforcement Learning (RL)

Reinforcement learning is a machine learning algorithm technique which is used to take up suitable actions in order to maximize the reward in a certain situation by learning the optimal behaviour in an environment. In the absence of a data set, a RL algorithm learns from experience [8]. In the concept of PCB design, RL can be used to automate certain aspects of the design processes. From the literature survey, it was observed that RL algorithms have been implemented for optimizing the power plane, the decaps in power delivery networks with the help of target impedance at the target IC ports [9]-[10].

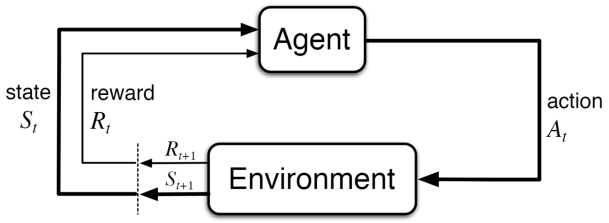


Figure 1.2: Block Diagram of Reinforcement Learning [2]

Figure 1.2 shows a few important elements of reinforcement learning. A brief description of these elements is provided below.

[2]Source: [11]

1. **Action:** Decisions made by the agent within an environment based on the current state is referred to as an action.

2. **Agent:** The agent is an entity which interacts with the environment, understands it and performs an action.

3. **Environment:** An environment is an external system with which the agent interacts through a set of actions.

4. **Policy:** Policy is the method that the agent follows in a time step after observing the environment.

5. **Reward:** Reward is a feedback that agent receives from the environment based on the actions chosen.

6. **State:** State is the current situation of the agent at a specific time. Each action leads to a change of state.

1.1.5 Multi-Agent Reinforcement Learning (MARL)

Multi-Agent reinforcement learning is a sub-branch of RL where multiple agents interact within an environment and learn to make decisions collectively to achieve a common goal or individual goals. Here, each agent learns a policy to map observations to actions in order to maximize the cumulative reward. The challenge in MARL lies mainly in the coordination and collaborations among the agents to achieve the optimal performance. Challenges such as credit assignment, decentralization decision-making and emergent strategies underscore the intricate nature of MARL.

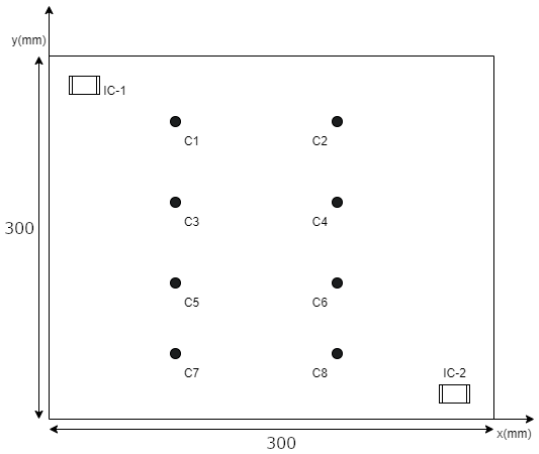
1.2 eCADSTAR

eCADSTAR is a comprehensive PCB design software that offers powerful tools for designing, simulating, and analyzing printed circuit boards (PCBs). With its intuitive interface and advanced features, eCADSTAR's PCB Editor provides engineers with a flexible and efficient platform to design complex PCB layouts. The software allows users to create schematic diagrams, route traces, place components, and perform design rule checks (DRC) to ensure compliance with industry standards. Additionally, eCADSTAR's PI EMI Analysis module enables engineers to analyze and optimize power integrity (PI) and electromagnetic interference (EMI) issues in their PCB designs. By simulating power distribution networks (PDN) and identifying potential noise sources, eCADSTAR helps designers mitigate risks and optimize performance, ultimately leading to more reliable and efficient electronic products [6].

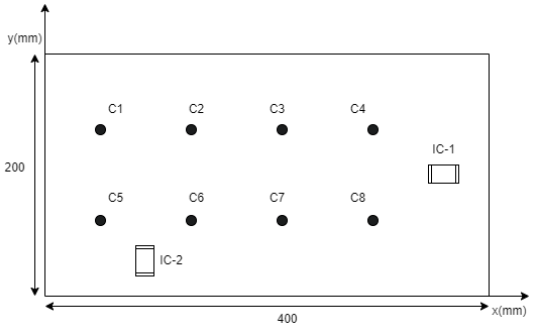
1.2.1 Design

The following PCB designs in Figure 1.3 were designed in the eCADSTAR PCB Editor and implemented for performing the analysis in the eCADSTAR PI EMI Analysis. The

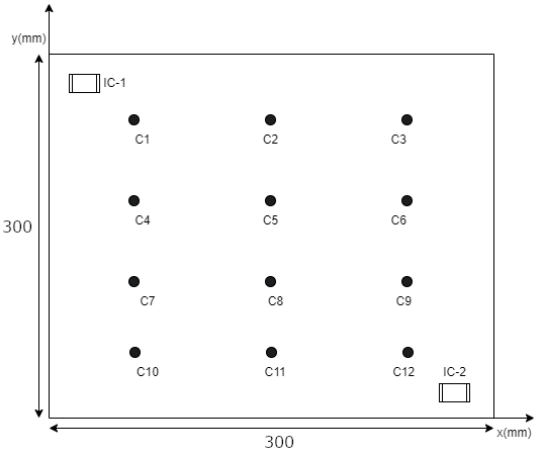
square boards have a dimension of 300mmx300mm while the rectangular shaped boards are measured as follows 400mmx200mm. Each of the four boards comprised of either 8 or 12 decaps with 2 ICs each. The frequency range to measure is set to be 2×10^5Hz to 2×10^8Hz and the impedance range that has been used is $7\,\Omega$ to $70\,\Omega$.



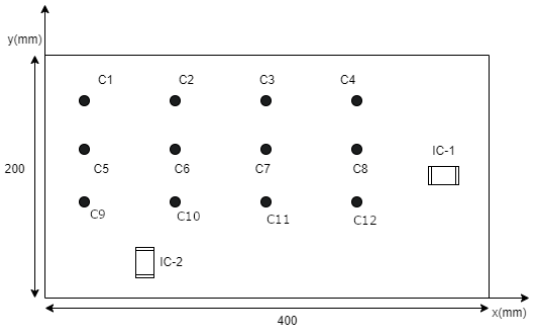
(a) Square Board with 8 Decaps



(b) Rectangular Board with 8 Decaps



(c) Square Board with 12 Decaps



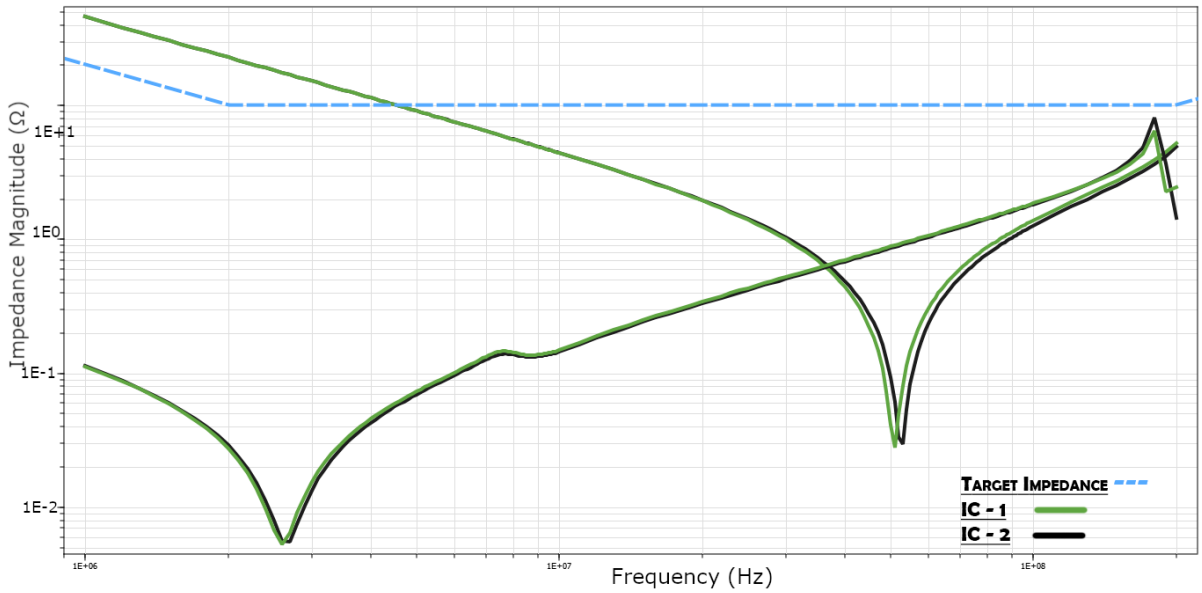
(d) Rectangular Board with 12 Decaps

Figure 1.3: Printed Circuit Boards implemented in the Project [3]

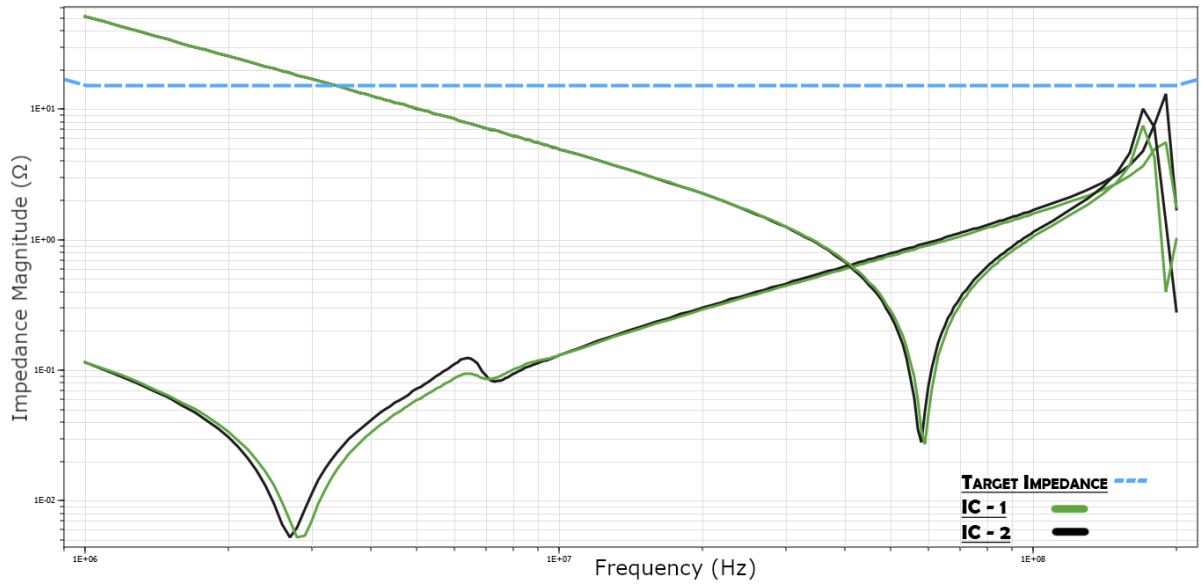
1.2.2 Impedance Profile

Figure 1.4 shows the Power Integrity (PI) analysis implemented on two different printed circuit boards. As seen in the figure, with respect to the target impedance, the number and the position of the capacitors is chose in the software such that the impedance profiles of the two ICs is maintained below the given target impedance. This software helps in analysing the optimal position of the ICs on the board. Thus, instead of doing the same manually, our project makes use of a MARL method to choose the optimal and efficient solution.

[3]Source: eCADSTAR PCB Editor 2023.0



(a) Impedance in db (Ω) for the Designed Square Board(12 ports)



(b) Impedance in db (Ω) for the Designed Rectangular Board(12 ports)

Figure 1.4: Impedance Profile for Two ICs [4]

[4]Source: eCADSTAR PI EMI Analysis 2023.0

2

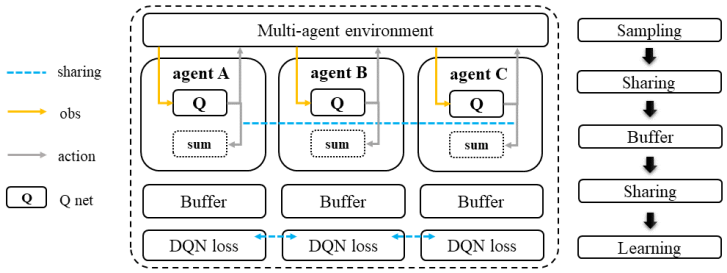
Multi-Agent Reinforcement Learning (MARL) Framework and Implementation

2.1 Qmix Algorithm for Multi-Agents

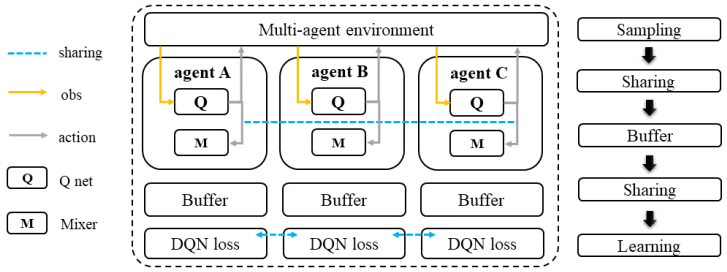
Multi-Agent Reinforcement Learning (MARL) algorithms are designed to tackle the challenges of environments where multiple agents interact and influence each other's outcomes. These algorithms can be categorized into several types based on the nature of agent interactions and objectives they aim to achieve. Cooperative MARL involves agents collaborating towards a common goal, while Competitive MARL sees agents with conflicting goals competing against each other. Mixed MARL encompasses scenarios where agents engage in both cooperative and competitive interactions, requiring dynamic balancing of cooperation and competition [22]. Centralized Training with Decentralized Execution (CTDE) approaches enable agents to train with global information while executing policies using only local observations. Learning Communication focuses on how agents share information to enhance learning and coordination. Understanding these types is essential for developing effective algorithms capable of navigating diverse multi-agent environments [12].

2.1.1 Cooperative MARL

Cooperative MARL involves agents collaborating towards a shared objective. Key methods in cooperative MARL include Value Decomposition Networks (Value Decomposition Networks (VDNs)), which decompose the joint value function into individual agent value functions. This enhances coordination while maintaining simplicity. QMIX, another method, builds upon VDN by incorporating a mixing network to combine individual value functions. This allows for more complex and effective coordination while ensuring a monotonic relationship with the joint value function. Q-value Transformed Reinforcement Learning (QTRAN) addresses limitations of VDN and QMIX by providing a flexible value decomposition approach, improving generalization and scalability. Additionally, Q-value Propagation Learning EXtension (QPLEX) utilizes a dueling network structure to further enhance scalability and performance in cooperative tasks [24]-[12].



(a) Value Decomposition Network



(b) Monotonic Value Factorisation

Figure 2.1: Difference in the Workflow [1]

Figure 2.1, shows a difference in the workflow of VDN and QMIX respectively. While each of the agents in VDN and QMIX follow the DRQN sampling pipeline, these algorithms differ in the following ways: In VDN, the Q-values and target Q-values of each agent and other agents are summed directly to obtain the joint Q-value. This summation process aggregates the contributions of all agents' actions to the environment without any additional processing. Whereas in QMIX, the Q-values and target Q-values of each agent and other agents are fed into a Mixer to obtain the overall Q-value of the team, denoted as Q_{tot}. Thus, involving an additional step of processing through a Mixer, which could incorporate more complex interactions or dependencies between agents' actions before generating the joint Q-value.

2.1.2 Competitive MARL

Competitive MARL involves agents with conflicting goals competing against each other. In this setting, game-theoretic approaches are often employed to find equilibrium strategies. These strategies ensure that no agent can unilaterally improve its outcomes by changing its strategy, thus fostering decision-making and adaptive behavior among competing agents [24].

2.1.3 Mixed MARL

Mixed MARL presents scenarios where agents engage in both cooperative and competitive interactions. This complexity necessitates a dynamic balance between cooperation and competition, challenging agents to develop advanced coordination strategies. Successfully

[1] Source: [14]

navigating mixed MARL environments requires agents to adapt their behavior based on the changing dynamics of cooperation and competition [24].

Figure 2.2: An Example of Mixed MARL [2]

Figure 2.2 shows an example of the implementation of Mixed MARL. It not only shows competitive but also cooperative MARL methodologies. As an example of cooperative MARL, the two agents in "purple" interact with each other in order to score a goal, same goes with the two agents in "blue". As far as competitive MARL is concerned, the agents in blue compete against those in purple to score the highest goal. Thus, illustrating how Mixed MARL incorporates both cooperative and competitive interactions among agents to achieve their objectives.

2.1.4 Centralized Training with Decentralized Execution (CTDE)

Centralized Training with Decentralized Execution (CTDE) is a methodology that trains agents with global information while allowing them to execute their policies using only local observations. A notable algorithm in this category is Multi-Agent Deep Deterministic Policy Gradient (Multi-Agent Deep Deterministic Policy Gradient (MADDPG)), which extends the Deep Deterministic Policy Gradient (DDPG) algorithm to multi-agent settings. MADDPG considers other agents' policies during training but executes independently, optimizing coordination and performance in complex environments [12].

[2]Source: [21]

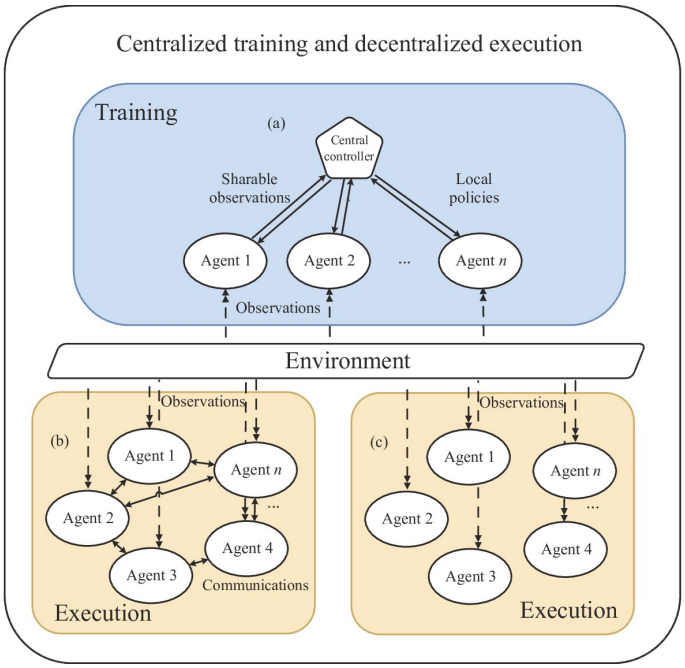


Figure 2.3: Framework of CTDE [3]

Figure 2.3 above depicts the structure of the CTDE method. Where, the "Training" represents the use of a centralized controller in order to control all agents. Further, "Execution (b)" is a realization of multi-agent control with limited communication. Finally, "Execution (c)" represents controlling each agent through individual controller, fully decentralized without communication.

2.1.5 Learning Communication

Learning Communication in MARL focuses on how agents share information to improve learning and coordination. This approach often involves the use of communication protocols or graph neural networks to facilitate information sharing among agents. By enhancing agents' ability to exchange information effectively, learning communication methods improve coordination and decision-making in dynamic and complex environments [24].

2.1.6 Why Qmix?

QMIX, compared to VDN, offers a more sophisticated approach in Cooperative MARL by incorporating a mixing network to combine individual value functions, allowing for better coordination and complexity management. Unlike QTRAN, which addresses limitations of VDN and QMIX but may lack the same level of effectiveness, QMIX maintains a balance between flexibility and performance [12].

[3]Source: [21]

In Competitive MARL, QMIX may not be directly comparable as it's designed for cooperative scenarios. However, its ability to effectively coordinate multiple agents towards a common goal could potentially provide an edge in competitive settings by fostering strategic collaboration. In Mixed MARL, QMIX's capability to handle complex cooperative interactions could be advantageous, although its effectiveness in balancing cooperation and competition would depend on the specific dynamics of the scenario [24].
Finally, in comparison to CTDE, QMIX's effectiveness in training cooperative agents could indirectly contribute to improved performance in CTDE setups by enhancing the agents' ability to coordinate during decentralized execution [12]. Overall, while QMIX may not necessarily be universally superior to all other algorithms in every context, its unique features make it a strong contender for addressing coordination challenges in Cooperative MARL scenarios [18].

2.1.7 Structure of Qmix

Figure 2.4 shows the structure of the Qmix algorithm. The following text gives an explanation of the same.

Mixing Network Structure:
The mixing network structure is a crucial component of QMIX. It is responsible for combining the individual agent values into joint action-value function Q_{tot}. The functionality of the structure is as follows, it takes in the outputs of the agent networks as its inputs which represent individual agent value functions $Q_a(\tau_a,\ u_a)$ and thus produces the joint action-value function Q_{tot}.

In order to ensure consistency between centralized and decentralized policies, the mixing network enforces a monotonic relationship between Q_{tot} and each of the Q_a. This constraint is crucial for tractable maximization of Q_{tot} during the off-policy learning. This structure is implemented as a feed-forward neural network by allowing complex non-linear combinations of individual agent values. The weights of the mixing network are constrained to be non-negative to enforce monotonicity. These weights are generated by separate hyper-networks. Each hyper-network takes the state s as its input and produces the weights of one layer of the mixing network. By conditioning the weights on the state, the mixing network can flexibly integrate extra state information into the joint action-value estimates.

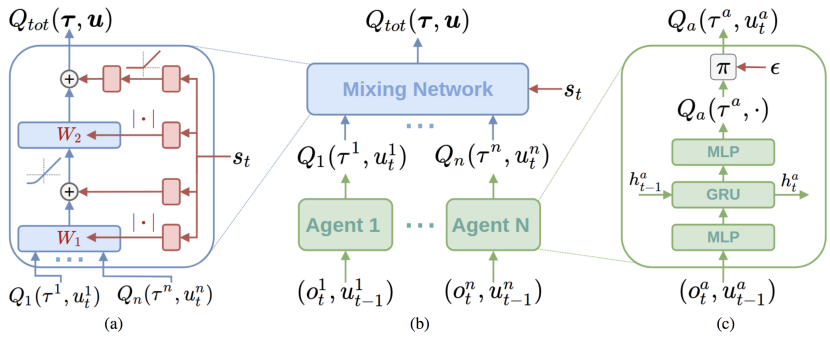


Figure 2.4: (a) Mixing Network Structure (b) Overall Qmix Architecture (c) Agent Network Structure [4]

Agent Network Structure:
The agent network is responsible for computing the individual agent value functions $Q_a(\tau_a, u_a)$. The functionality of the network goes as such, the agent network for each agent a takes the current individual observation o_a^t and the last action u_a^{t-1} as its input and outputs the value function $Q_a(\tau_a, u_a)$. This network is mainly implemented as a Deep Recurrent Q-Network (DRQN) thus allowing it to handle sequential data efficiently. It consists of recurrent layers which process the temporal dynamics of the agent's environment and fully connected layers that process the observation and action inputs. At each time step, the structure receives the current observation o_a^t and the last action u_a^{t-1} as the input. This allows the network to capture the sequential nature of the agent's interactions with the environment. The output of the agent-network is the value function $Q_a(\tau_a, u_a)$ which represents the expected cumulative reward that an agent a can achieve by following the policy a and taking the action u_a.

Overall Qmix Architecture:
This architecture combines the mixing network and the agent networks to form a unified framework for cooperative MARL. For the integration, each agent has its own agent network that computes its individual value function and these values are further combined by the mixing network to compute the joint action-value function Q_{tot}. The architecture enforces a monotonic relationship between Q_{tot} and each of the Q_a, thus ensuring consistency between the centralized and the decentralized policies. The entire QMIX structure is trained end-to-end to minimize the loss function, which is analogous to the standard Deep Q-Network (DQN) loss. This allows the network to learn to approximate the optimal joint action-value function Q* efficiently.

2.1.8 QMix Framework

Qmix operates by estimating joint action-values through a neural network. This leverages the collective insights from individual agent's local observations to compute a complex combination of per-agent values. It ensures that the joint action-value remains monotonic. Thus, enabling straightforward maximization of the joint action-value during off-policy learning. This structural enforcement guarantees alignment between centralized training

[4]Source: [15]

and decentralized execution, ensuring consistency across the board. Below mentioned Algorithm 2.1.1 describes the working of Qmix.

Algorithm 2.1.1.: QMIX Algorithm [5]

Initialize agent networks with parameters θ_i for each agent i
Initialize mixing network with parameters ϕ
Initialize hyper-networks that generate positive weights for the mixing network
repeat
    for each agent i **do**
        Observe local observation o_i
        Select action a_i based on policy derived from $Q_i(o_i, \cdot; \theta_i)$
    Execute actions $\mathbf{a} = (a_1, \ldots, a_n)$, observe reward r and next observation $\mathbf{o}'$
    For each agent i, update Q_i using local experience (o_i, a_i, r, o_i')
    Calculate target values for each agent Q_i'
    Use hypernetworks to update weights of the mixing network
    Update ϕ by minimizing the loss:

$$L(\phi) = \left(r + \gamma \max_{\mathbf{a}'} Q_{\text{tot}}(\mathbf{o}', \mathbf{a}'; \phi') - Q_{\text{tot}}(\mathbf{o}, \mathbf{a}; \phi) \right)^2$$

until convergence

2.2 Library

A library in the context of software development is a collection of precompiled routines that a program can use. These libraries provide functionality that can be easily reused across different programs, saving time and effort in software development. Libraries are essential for simplifying complex tasks and promoting code reuse, ensuring that developers do not need to rewrite common functionalities. Ray is an open-source framework designed to scale AI and Python applications, easing the complexity of machine learning workflows through scalable libraries, distributed computing primitives, and integration utilities for deploying a Ray cluster. Ray Reinforcement Learning Library (RLlib) is highlighted for its capability to handle scalable, distributed reinforcement learning. The framework comprises three layers: Ray Artificial Intelligence (AI) Libraries, Ray Core, and Ray Clusters. Specifically, the Ray AI Libraries and RLlib allow for the training and optimization of policies, management of multi-agent interactions, and provide functions like train(), compute_action(), save(), and restore() to manage the reinforcement learning models [17].

2.2.1 Ray RLlib

The main library used in the code for the implementation of MARL is the Ray RLlib. Ray is an open-source framework for scaling AL and Pyhton applications. Ray minimizes

[5] Source: [15]

the complexity of machine learning workflows with the help of scalable libraries, pyhtonic distributed computing primitives and integration and utilities for integrating and deploying a Ray cluster [17].

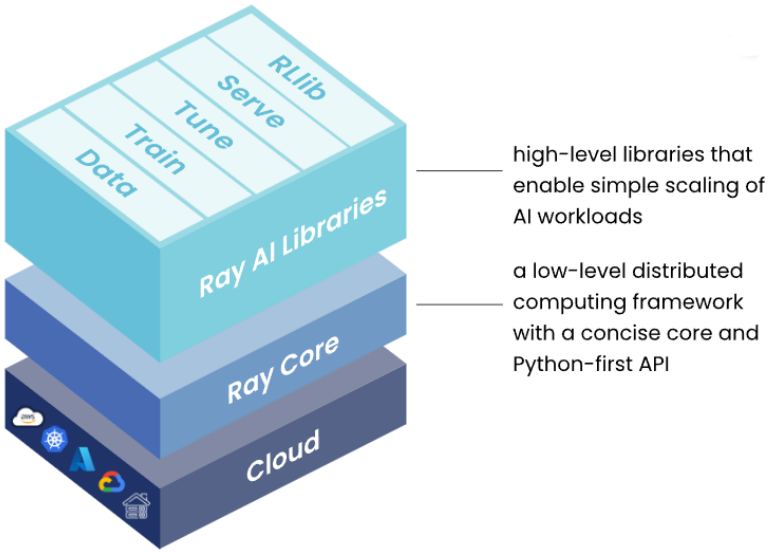


Figure 2.5: Stack of Ray Libraries [6]

The image above, Figure 2.5, depicts Ray's unified framework which comprises of the following three layers: Ray AL Libraries, Ray Core and Ray Clusters. We will focus on the Ray AL Libraries to give a brief on RLlib. RLlib is a scalable distributed reinforcement interaction. Through the algorithm's interface, the policy can be trained to compute actions or store the algorithms. In multi-agent training,, the algorithm manages the querying and the optimization of multiple policies at once [16].

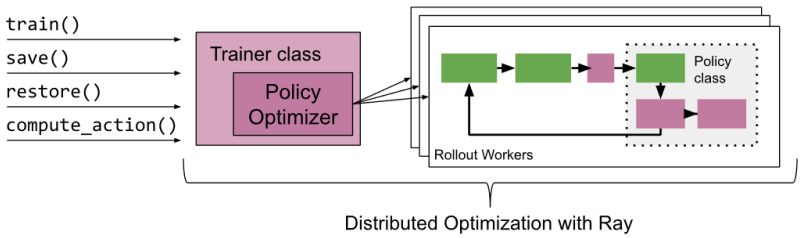


Figure 2.6: Structure of RLlib [7]

Figure 2.6 shows the structure of RLlib in Ray.

1. **train():** The function is used to update the RL model's parameters based on the collected data.

2. **compute_action():** The function calculates the action for a given state using the current policy.

[6]Source: [17]
[7]Source: [16]

3. **save() and restore():** These functions are used for persisting and loading the RL model's state, allowing for check-pointing and reusing trained models.
4. **Policy:** In RL, a policy defines the agent's behavior at each time-step. It can be a function, a network, or a class, depending on the implementation.
5. **Trainer class:** This is a class which includes methods for training, saving, and restoring the model.
6. **Optimizer:** The optimizer is used to update the RL model's parameters during the training process.
7. **Rollout workers:** Rollout workers are processes that collect data by interacting with the environment.

3

Environment and Implementation

3.1 Environment and Implementation

Our implementation is based on the structure of the RLlib library, which uses the Qmix algorithm. The first step to solve the problem is to choose the version of RLlib and the other libraries with which it generates dependency. This is because, with the passage of time, there are updates in the libraries, and some functionalities are eliminated or changed their function. For this, we chose the most stable version of the algorithm which is in the 2.5.1 version of RLlib with Python 3.8. Ray lib allows us to work with TensorFlow and Torch at the same time in some algorithms; unfortunately, this is not the case with Qmix, which can only use Torch. Implementing the algorithms using these libraries can be roughly divided into two parts: the environment and the configuration. The parts of the code and how we implement them are explained in detail below.

3.1.1 Environment

The main challenge of the implementation was the development of the environment for the analysis of the PCB, configuring the actions, and the observation spaces. The ray library environment must have a clear structure and key elements for its operation. The basic structure is based on initialization, reset, and step. Inside the initialization are all the variables necessary for the process to be carried out. This part also contains the key parts of the environment.
The designed environment effectively models the multi-agent problem of decap placement on a PCB. Key aspects include:
Efficient State Representation: Binary vectors allow for clear and concise state representation.
Reward Function: Tailored to encourage achieving target impedance while minimizing decap usage.
Step Function: Ensures correct transitions and environment interactions, including resetting and updating PCB configurations.

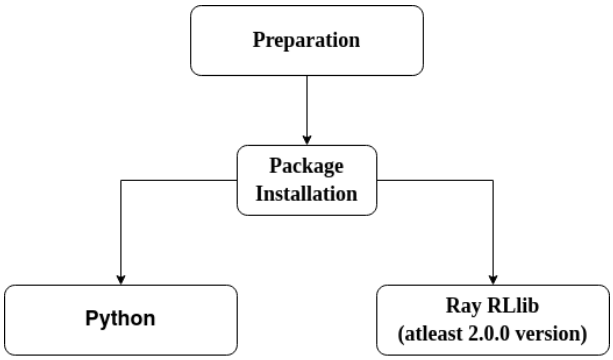


Figure 3.1: The Environment Setup [1]

After setting the action and observation space, we proceed with the reset function, where all important variables in a step function return to their initial state, such as the state and the PEB file used to perform the simulation in each iteration. Within the step function lies the process to be carried out for each agent; in this case, an action is chosen, which is then written into the PEB file, and the simulation is executed. In our approach, we chose to use two actions per step, each for one agent. Afterwards, the simulation is run only once, and the results are saved in two different CSV files, one for each IC. These are used to calculate the impedance corresponding to each IC, and with this comparison, we assign a corresponding reward. After setting the individual environment, we create the wrapper for the multi-agent environment, where we set a group of action and observation spaces for each agent and a bypass to the step and reset function of the single agent. 3.1 gives an overview of the custom environment.

3.1.2 Observation and action space

The key point for implementing the RLlib algorithms is the correct design of the action and observation spaces, which the algorithm uses to calculate the Q values. For the design of the action space, we encountered the limitation that the algorithm is designed only for discrete actions. Initially, we intended to use vectors with two positions to represent the type and position of the capacitor, but this was not possible. Therefore, we had to design a new strategy to implement the actions. We created a static list of possible actions and designed a discrete action that we use as an index to access the actions. This approach allows us to select a maximum of 24 different actions, which is the maximum value that can be used in discrete variables.

On the other hand, for the design of the observation space, we used a vector representing the positions of the capacitors on the PCB. The vector has N+1 elements, where N is the number of ports on the board, and the extra element represents the agent to which this space belongs. the format of the observation space is the following:

[1]Source: [13] was used for designing the flowchart

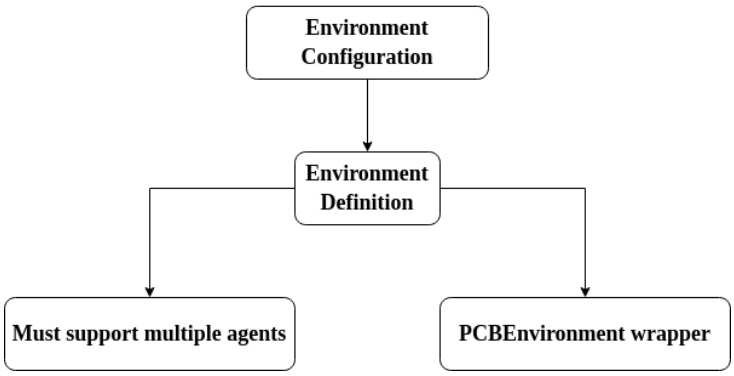


Figure 3.2: Environment and Implementation [2]

$$\begin{bmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & A \end{bmatrix}$$

Explanation: We decided to represent our observation spaces as vectors with 8 and 12 elements, representing the possible positions of the decaps on the PCBs, the values ranging from 0 - 2 where 0 represents the absence of capacitors and the others the value of the capacitors in the case of the 8-position tests.

The observation space offers a remarkable flexibility. Vectors can be used to represent our state, and it's not limited by the number of possible actions. However, it's still necessary that each element of the vector is a discrete variable, so we use the multi-discrete variable. Figure 3.2 gives a glimpse on the points kept in mind to design the environment for our project. Since out project focuses on the MARL concept, it was very important to design the environment by ensuring it runs not just for optimizing the position of the decaps for one IC but for many ICs. Apart from that we also had to ensure that the environment is compatible with the Ray Rllib library.

3.1.3 Configuration for the Library

The configuration is the most important thing when using an algorithm using Ray Lib. From version 2.0, the configuration is handled as a dictionary with the parameters needed to set the training and running of the code; the main parameters we can find are the epsilon values, the number of iterations, and the termination criteria. As well, for multi-agent Qmix, it is necessary to register the environment with the action and states in the form of a group of dictionaries. Figure 3.3 shows the working on how the configuration of the Qmix algorithm for the library works.

[2]Source: [13]

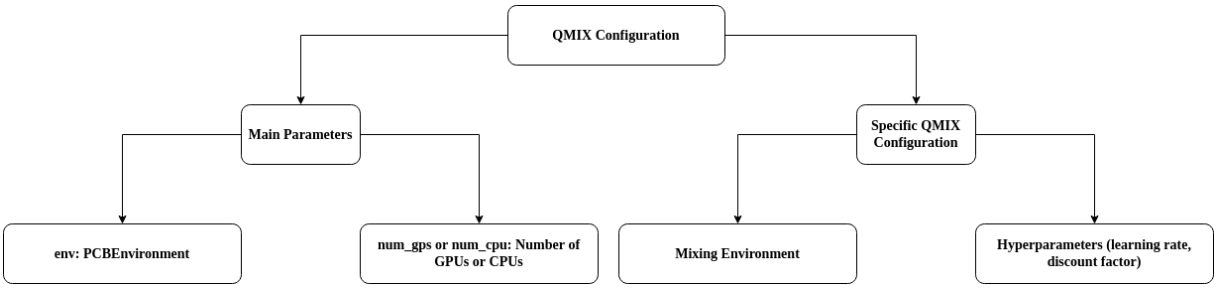


Figure 3.3: Qmix Configuration [3]

3.1.4 Execution and Results

At the conclusion of each iteration, the program's results are meticulously recorded in a comprehensive document. This document not only captures the reward values for each iteration but also provides a detailed analysis, including the average change in rewards, their maximum and minimum values. Such detailed tracking allows for a thorough evaluation of the program's performance over time.
Moreover, the document includes critical information about the time taken to complete each iteration, offering insights into the efficiency and speed of the program. It also highlights the most frequently used actions, shedding light on the strategies that the program relies on the most. This information is invaluable for understanding the program's decision-making process and for identifying areas that may require optimization or adjustment.
In addition to this, we generate a separate file that meticulously stores the state at which the termination criterion is met for each iteration. This file is crucial for understanding the conditions under which the program concludes its operations, providing a clear picture of its stopping points and helping to refine termination conditions for future runs.

[3]Source: [13]

4

Results and Evaluation

This chapter focuses on the results that were evaluated on a set of printed circuit boards. The evaluation process involved rigorous testing and analysis to assess the effectiveness of the proposed Multi-Agent Reinforcement Learning (MARL) methods for decaps placement. The primary objective was to determine how well the MARL approach performed in terms of reducing the number of decaps used while improving the overall performance of the PCBs.

Computer	RAM	CPU Cores	Clock Cycle (GHz)
1	32GB	14	3.8
2	32GB	12	2.2
3	32GB	16	3.8

Table 4.1: Specifications of the PCs used for the Project

We conducted comprehensive tests in four different scenarios, utilizing the PCBs shown in the image 1.3. In both cases, 12 and 8 capacitors were considered. In the tests with 12 ports, only one type of capacitor was used, while in the case of 8 capacitors, we run the test for a single type of capacitor and the case with two types of capacitors, ensuring a thorough examination of the model's performance.

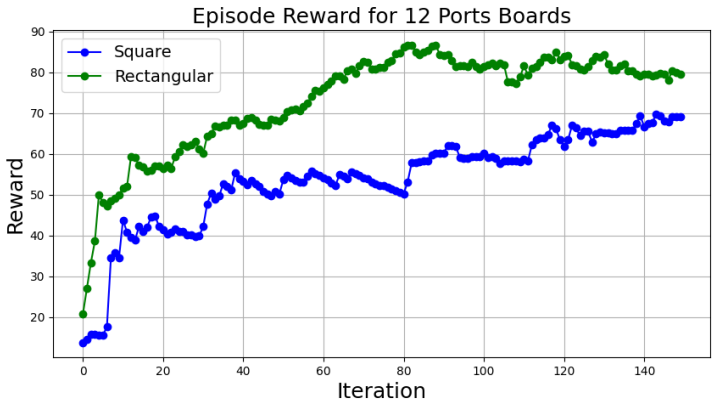
The simulation of the model was performed on 3 different computers, the specifications of which can be found in the table 4.1. This was done to check the fidelity of the algorithm depending on the capacity of the computer. The computer specifications include details such as the processor type and speed, the amount of RAM, and the type of graphics card, all of which can significantly affect the performance of the algorithm.

The first thing that was observed was that the time that the program takes to be processed is very long, and during the first tests we realized that we had to maintain a minimum number of iterations to be able to observe a good performance of the model. This is because the model needs a certain number of iterations to learn and adjust its predictions based on the feedback it receives.
The average time that the program takes to run is 15 hours, but the exacts times are shown in the tables of the section 4.2. Of the total time 70% is the simulation within

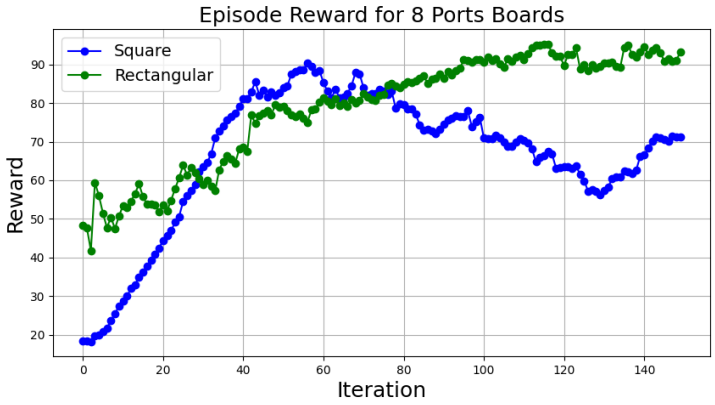
the Zuken program and the remaining time is the processing of the actions and rewards of the algorithm. The rllib libraries allow us to use GPU to process the program which increases the speed of the processing of Q values calculation, but the simulation time in the analysis program remains the same, so it does not make a big difference.

4.1 Episode Reward

In the graphs 4.1, we can see the behavior of the main reward when running the code on different board configurations; we can observe that on each test, the rewards have a similar behavior, but the most notorious difference is the reward values, considering that in each iteration, the code is compiled at least 15 times. The reward is added in each iteration, then each computer and test follows different paths.



(a) PCB Board with 12 ports and one type of decap



(b) PCB Board with 8 ports and one type of decap

Figure 4.1: Graphs of the Mean Reward per Iteration

In 4.1a you can observe the variations in the rewards for the rectangular and the square boards with 12 decaps each. Throughout the iterations, it can be seen that the rewards

earned by the rectangular board are comparatively higher than the square board . Although, the behavior of both the boards indicates a system approaching steady state, the overall result for the rectangular board seems to be better than the square board.

For the graph in 4.1b, again the rewards for the rectangular board seem to be much better that the square board. Here, the boards used have 8 decaps instead of 12. The square board shows a significant rise in its values, but half way through the program, the reward values start to show a dip in the curve. Yet again, due to the higher rewards of the rectangular board along with the steadiness of the graph, it is considered to have a better output than the square board.

In conclusion, since the same set of hyperparameters have been used for each of the 4 boards, fine tuning these parameters for the square boards could yield better and promising results.

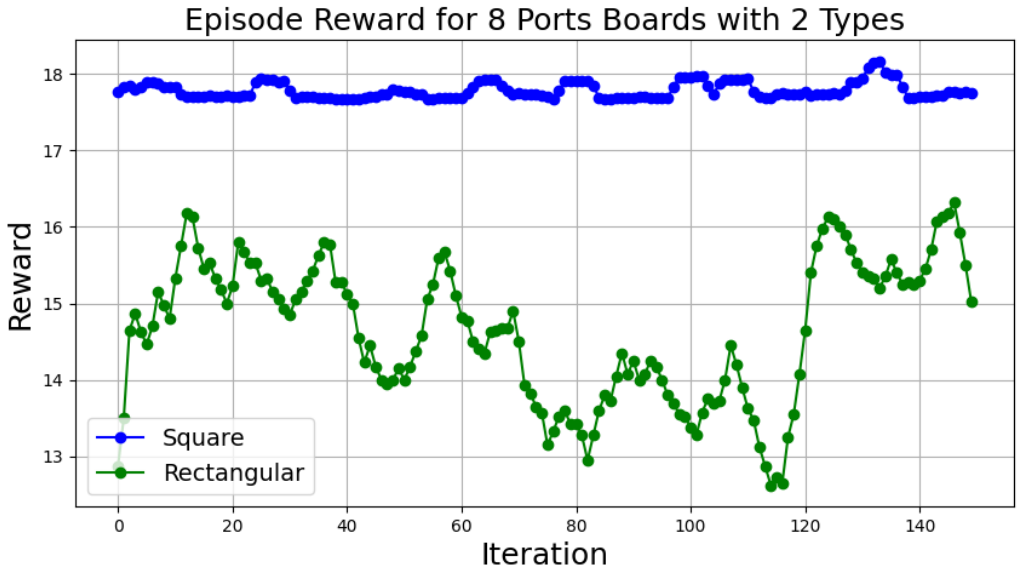


Figure 4.2: PCB Board with 8 ports and two type of decaps

4.2 shows a unique set of results: Both the boards comprise of 8 decaps but use two different types of these decoupling capacitors. The rectangular board shows inconsistency in the values obtained. The continuous rise and fall depicts the instability in attaining an optimal result. On the other hand, the values for the square board show an almost static result. Compared to the previous graphs of the boards using only one decap, the reward values too are at a lower end. These results indicate that the algorithm is struggling to choose the right set of actions to improve the performance.

To perform these tests, we used the same hyperparameters as in the cases with a single type of capacitor, expecting to have similar results. However, as can be observed, the system's behavior is irregular and fails in the learning task. To correct this, it is necessary to modify the learning rate values and possibly increase the batch size. As seen in the figure 4.6d in the loss section, the system completely fails, never attempting to explore

new actions and always sticking with the same ones.

4.2 Optimal placement

The following tables and figures show the best results obtained in each test, showing the optimal positions that were repeated the most during the reinforce learning process. As can be observed, in most cases, computer 1 presents a greater number of repetitions of the optimal positions. This is because it was our main computer in which more tests were made, and the libraries of rllib use the previous training for each new simulation, allowing us to converge on the optimal solution quickly.

Computer	Train Time	Optimal Layout	Optimal Layout											
			1	2	3	4	5	6	7	8	9	10	11	12
Computer 1	47h11m33s	316	0	1	0	0	0	0	0	0	1	0	0	0
Computer 2	14h5m22s	4	1	0	0	0	1	1	1	0	0	0	0	0
Computer 3	14h57m6s	4	1	0	0	0	0	1	1	0	1	0	0	0

Table 4.2: Optimal Layout Results 12 Ports Square Board

Computer	Train Time	Optimal Layout	Optimal Layout											
			1	2	3	4	5	6	7	8	9	10	11	12
Computer 1	19h6m8s	12	0	1	0	0	1	0	0	0	0	0	1	0
Computer 2	23h56m46s	7	0	1	0	0	0	1	0	1	0	0	0	0
Computer 3	24h33m6s	7	1	0	0	0	0	0	0	0	0	0	0	1

Table 4.3: Optimal Layout Results 12 Ports Rectangle Board

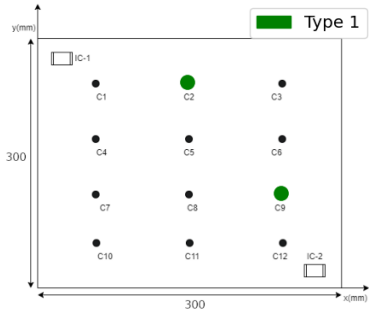
Computer	Train Time	Optimal Layout	Optimal Layout							
			1	2	3	4	5	6	7	8
Computer 1	12h21m42s	22	0	0	1	0	0	0	1	0
Computer 2	14h5m22s	38	0	0	0	1	0	0	2	0
Computer 3	12h8m50s	224	0	0	0	0	0	0	0	1

Table 4.4: Optimal Layout Results 8 Ports Square Board

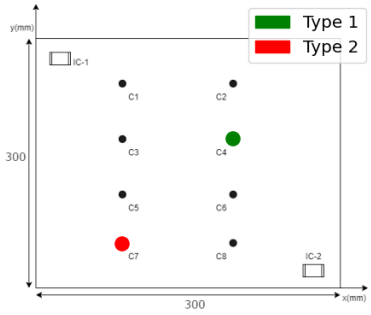
Computer	Train Time	Optimal Layout	Optimal Layout							
			1	2	3	4	5	6	7	8
Computer 1	12h17m33s	333	2	0	0	0	0	1	0	0
Computer 2	14h4m22s	102	0	0	0	0	0	0	2	0
Computer 3	12h12m22s	284	0	1	0	0	0	0	1	0

Table 4.5: Optimal Layout Results 8 Ports Rectangle Board

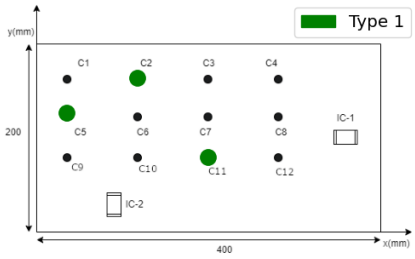
As observed in Table 4.4, in the case of the PCB with 12 ports, an average of 2 capacitors are needed to achieve the target impedance. However, there were also results where the target could be met by placing one capacitor, but this occurred less frequently compared to combinations with two capacitors.
On the other hand, for PCBs with 8 ports and two types of capacitors, there is a tendency to use type 1 capacitors. Similar to the previous case, the desired impedance can be achieved using 2 capacitors.



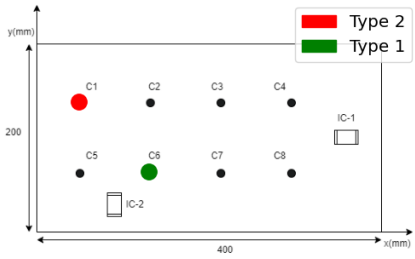
(a) Optimal collocation in the 12 Port Square Board



(b) Optimal collocation in the 8 Port Square Board



(c) Optimal collocation in the 12 Port Rectangular Board



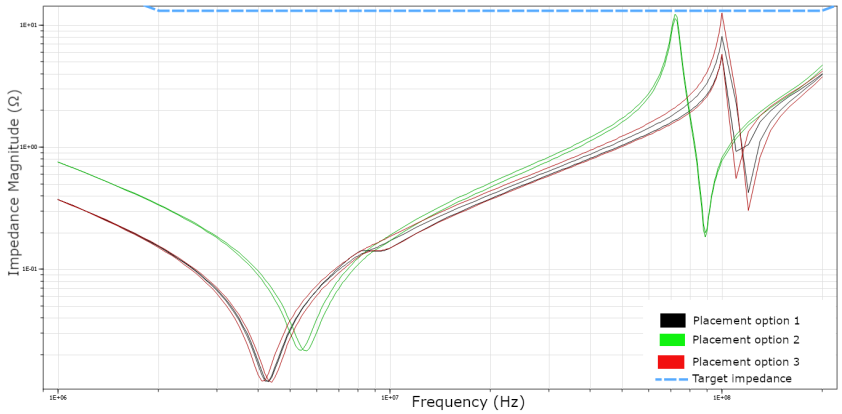
(d) Optimal collocation in the 8 Port Rectangular Board

Figure 4.3: Optimal Position on the boards

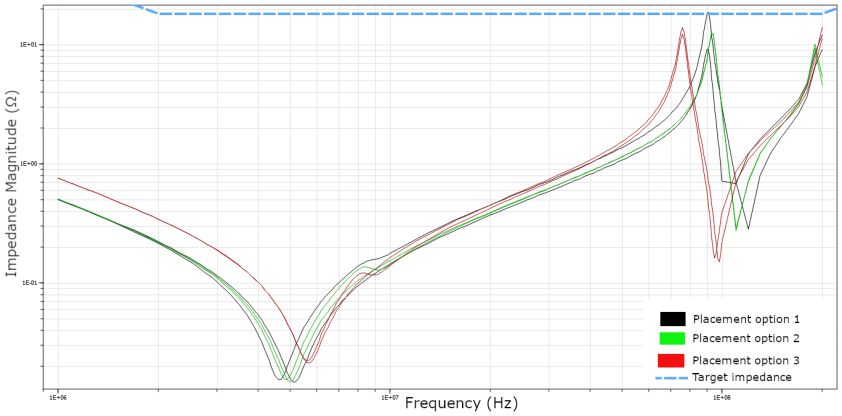
Logically, we expected that the optimal positions would be those near the ICs, or in the case of the square board, the capacitors located in the middle, as these would complement each other to reduce the impedance of both ICs simultaneously.However, this is different from what we found in the results.
The positions of the capacitors on the board can be observed in the 4.3, where the green color signifies that a type 1 capacitor is required in that position, and the red color represents type 2. 4.5a shows the optimal result being the capacitors close to each of the ICs. Further, for 4.5b, the decaps of each type, appear to be closer to IC2 as compared to IC1.

Moreover, for 4.4a, 3 capacitors have been chosen in order to obtain an optimal result. Again, these capacitors seem to be closer to IC2. Finally, 4.4b, with two types of decaps, gives an optimal output for 2 capacitors of each kind at positions closer to IC2.

With the positions selected on the boards, the PI analysis program by Zucken was used to perform the impedance simulations to verify that the solutions provided by the algorithms meet the target impedance we imposed. The results can be seen in the Figure 4.4



(a) Square Board with 12 ports

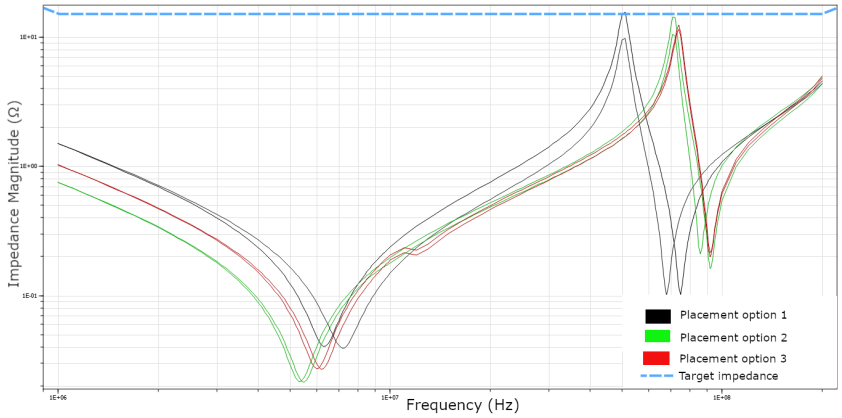


(b) Rectangular Board with 12 ports

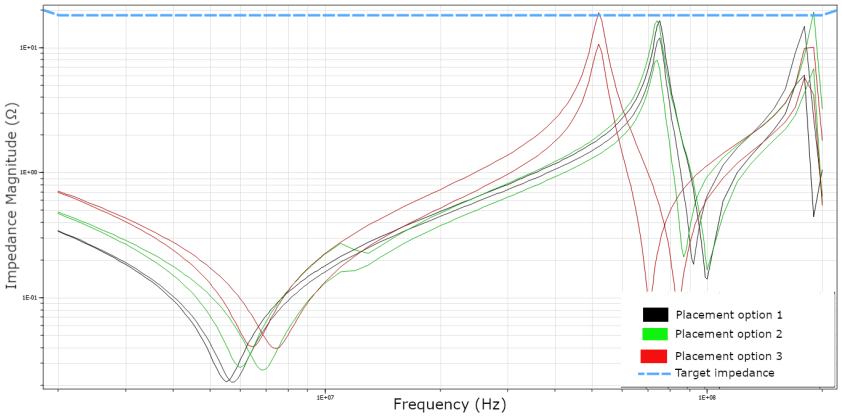
Figure 4.4: Simulation of the Optimal Results 1 Type of Capacitor [1]
(Black:Computer 1, Green:Computer 2, Red:Computer 3)

The four graphs in 4.3 and 4.4, obtained via simulation in eCADSTAR PI and EMI analysis, show that the impedance levels were maintained below the threshold. This proves that the algorithm was successful in choosing the right set of actions most of the times. Although the graphs for the 8 ports seem to be maintained almost below the threshold, the episode reward and the loss graphs show instability in attaining the same. Once the hyper-parameters are well adjusted, the PI analysis of these boards would give better results. In order to get the above results, the optimal options of the placement and/or the types of decaps were used to perform the simulations.

[1]Source: eCADSTAR PI EMI Analysis 2023.0



(a) Square Board with 8 ports



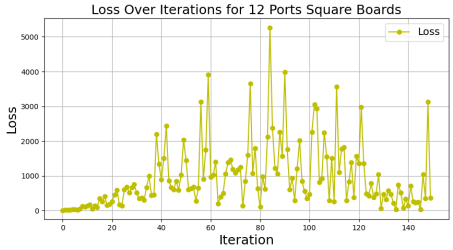
(b) Rectangular Board with 8 ports

Figure 4.5: Simulation of Optimal Results 2 Types of Capacitors [2]

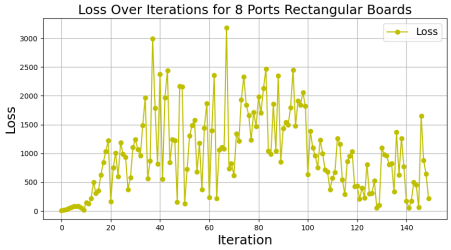
4.3 Loss

The loss over iteration graph allows us to observe the model's performance and learning over time by measuring how well the model predictions resemble the target values and calculating the error between the predicted and actual values.
High loss values mean the error between the prediction and the actual value is significant. In this type of graph, the loss value is expected to be constant and close to zero. This behavior indicates that the model performs well and that the learning task is effective because it suggests that the model's predictions are consistently close to the actual values, with minimal error.

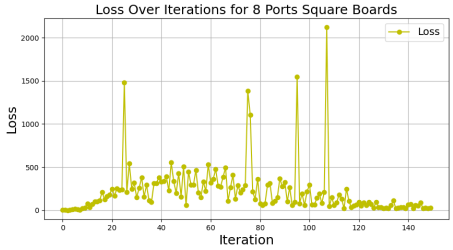
[2]Source: eCADSTAR PI EMI Analysis 2023.0



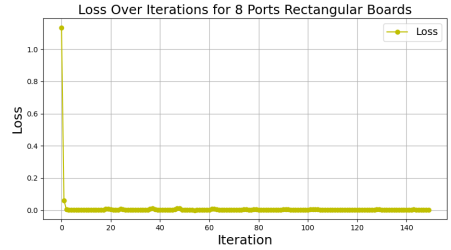
(a) Square Board with 12 ports



(b) Rectangular Board with 12 ports



(c) Board with 8 ports 1 Type of Decap



(d) Board with 8 ports 2 Types of Decaps

Figure 4.6: Loss over Iterations

The figure 4.6a shows fluctuations in the error values with the highest loss value going above 5000. These fluctuations indicate a difference in predicting the actual values. Similarly,graph 4.6b, shows significant fluctuations in the loss throughout. Although, despite these large loss values for both the graphs, towards the end, the loss reduces to a great extent, thus indicating that the algorithm works towards attaining an optimal result.

In the figure 4.6c, we see that the fluctuations are quite less, with a few outliers, else the loss remains within the range of 0 to 500. This depicts the algorithms prediction in choosing the actual values. A better graph for the least amount of loss would be, 4.6d, which is for a board with 8 ports and two types of decaps. The loss is almost zero with the highest loss being 1. Thus, for this board, the algorithm could easily predict the actual values.

As mentioned at the beginning, we would expect our loss graph to remain with low and stable values, but the discrepancy in the values indicates that the model is experimenting with new actions. This allows it to learn new actions that lead to the optimal path, as seen in the case of our design with two capacitors shown in the figure 4.6d. The behavior of the loss seems ideal because the expected values are the same as the measured ones. However, this indicates that the model never underwent a learning process, so a stable state could not be achieved in these tests.
In the other cases, it is observed that the highest peaks of loss occur in the middle of the process, and a more stable state is found at the beginning and end.

5
Conclusion

In conclusion, our project demonstrates the significant potential of utilizing the QMIX algorithm to optimize the placement of decoupling capacitors on printed circuit boards (PCBs). The application of this algorithm on the board with 12 capacitors yielded promising results, as it reached a stable state after 150 iterations. This stability indicates effective learning and successful optimization of capacitor placement, highlighting the algorithm's capability in this context. Conversely, for the board with 8 capacitors of two different capacitor types, the reward did not converge as expected. This discrepancy suggests that the current set of hyper-parameters may not be ideal for configurations involving capacitors of varying values.

These findings emphasize the necessity for further research and adjustment of hyper-parameters to enhance the algorithm's performance in such diverse configurations. Future work will be dedicated to fine-tuning these parameters to achieve better convergence and optimization outcomes. Moreover, exploring alternative machine learning techniques or hybrid approaches could offer additional improvements. Overall, our methodology demonstrates considerable promise in automating PCB design processes and improving the reliability of electronic circuits. With continued refinement and optimization, this approach has the potential to significantly reduce design time and increase the efficiency of PCB manufacturing.

A suggested future application of this research could be the use of the QMIX algorithm in the design and optimization of flexible PCBs (FPCBs) and rigid-flex PCBs, which are increasingly used in advanced electronics such as wearable devices, foldable smartphones, and medical implants. These types of PCBs often require more complex design considerations due to their unique physical properties and operational environments. Integrating the QMIX algorithm could help in optimizing the placement of decoupling capacitors and other components to ensure high performance, reliability, and durability in these innovative applications. This could lead to enhanced device functionality and longevity, pushing the boundaries of what is achievable with modern electronic designs.

Bibliography

[1] **ARROW:** *Why decoupling capacitors matter.* Available at: https://www.arrow.com/en/research-and-events/articles/why-decoupling-capacitors-matter. 2023.

[2] **AUTODESK:** *What Are Decoupling Capacitors?* 2024.

[3] **H. G. Baoyin Hua and N. Mittal:** *Optimization of power delivery network using Reinforcement Learning Algorithms.* 2023.

[4] **E. Bogatin:** *Signal and Power Integrity - Simplified.* 2nd. Pearson Education, 2009.

[5] **Cadence:** *Decoupling Capacitor Placement in PCB Layout.* Available at: https://resources.pcb.cadence.com/blog/2020-decoupling-capacitor-placement-in-pcb-layout.

[6] *eCADSTAR PCB Design Software.* https://www.ecadstar.com/. Accessed: May 2024. Zuken.

[7] **S. ENGINEERING:** *Power Delivery Network (PDN).* Available at: https://semiengineering.com/knowledge_centers/low-power/low-power-design/power-delivery-network-pdn/.

[8] **G. for Geeks:** *Reinforcement learning.* Available at: https://www.geeksforgeeks.org/what-is-reinforcement-learning/. 2023.

[9] **S. Han, O. W. Bhatti, and M. Swaminathan:** "Reinforcement Learning for the Optimization of Decoupling Capacitors in Power Delivery Networks". In: *2021 IEEE International Joint EMC/SI/PI and EMC Europe Symposium.* 2021.

[10] **S. Han, O. W. Bhatti, and M. Swaminathan:** "Reinforcement Learning for the Optimization of Power Plane Designs in Power Delivery Networks". In: *2022 IEEE 31st Conference on Electrical Performance of Electronic Packaging and Systems (EPEPS).* 2022.

[11] **J. Hu, S. Wang, S. Jiang, and W. Wang:** "Rethinking the Implementation Tricks and Monotonicity Constraint in Cooperative Multi-agent Reinforcement Learning". In: *ICLR Blogposts 2023.* https://iclr-blogposts.github.io/2023/blog/2023/riit/. 2023. URL: https://iclr-blogposts.github.io/2023/blog/2023/riit/.

[12] **D. Huh and P. Mohapatra:** *Multi-agent Reinforcement Learning: A Comprehensive Survey.* 2023. arXiv: 2312.10256 [cs.MA].

[13] **JGraph:** *Flowchart Design Tool*. https://www.drawio.com/. Accessed: 2024-05-30.

[14] **MARLLib Development Team:** *MARLLib Documentation: Joint Q Family Algorithms*. https://marllib.readthedocs.io/en/latest/algorithm/jointQ_family.html. 2023.

[15] **Priyank:** *Multi-Agent Reinforcement Learning*. https://ppriyank.github.io/MARL/final.html. 2019.

[16] **RAY:** *Getting Started with RLlib*. Available at: https://docs.ray.io/en/latest/rllib/rllib-training.html#getting-started-with-rllib.

[17] **RAY:** *Ray: An open-source framework for scaling AI and Python applications*. Available at: https://docs.ray.io/en/latest/ray-overview/index.html. Accessed: 2024-05-19.

[18] **P. K. Sharma, E. G. Zaroukian, R. Fernandez, A. Basak, and D. E. Asher:** "Survey of recent multi-agent reinforcement learning algorithms utilizing centralized training". In: *Artificial Intelligence and Machine Learning for Multi-Domain Operations Applications III*. Apr. 2021. URL: http://dx.doi.org/10.1117/12.2585808.

[19] **M. Swaminathan and E. Engin:** *Power Integrity Modeling and Design for Semiconductors and Systems*. Prentice Hall, 2007.

[20] **A. TECHNICAL:** *The Importance of Printed Circuit Boards for Electronics*. Avai;able at: https://www.apollotechnical.com/importance-of-printed-circuit-boards-for-electronics/. 2021.

[21] **J. Wang, Y. Hong, J. Wang, J. Xu, Y. Tang, Q.-L. Han, and J. Kurths:** *Cooperative and Competitive Multi-Agent Systems: From Optimization to Games*. 2022. URL: %5Curl%7Bhttps://www.ieee-jas.net/en/article/doi/10.1109/JAS.2022.105506%7D.

[22] **K. Zhang, Z. Yang, and T. Başar:** *Multi-Agent Reinforcement Learning: A Selective Overview of Theories and Algorithms*. 2021. arXiv: 1911.10635 [cs.LG].

[23] **R. Zhang, Y. Cao, Q. Huang, and B. Yu:** "Reinforcement Learning-Based Power Delivery Network Optimization for System-on-Chip Designs". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 39.10 (2020), pp. 2107–2118.

[24] **Z. Zhou, G. Liu, and Y. Tang:** *Multi-Agent Reinforcement Learning: Methods, Applications, Visionary Prospects, and Challenges*. 2023. arXiv: 2305.10091 [cs.AI].

6 Appendix

Board	Hyper-parameter					
	Mixing embed	Learning rate	Batch size	Epsilon time steps	Optimal alpha	Optimal epsilon
12-ports Board	32	0.0005	32	40	0.99	0.00001
8-ports Board	32	0.0010	64	40	2	0.00002

Table 6.1: Hyper-parameters for Different Boards

6.1 Python Code

6.1.1 Policies and Observation Space for Qmix

```
#########
    grouping = {
        "group_1": [0, 1],
    }
    obs_space = Tuple([
        Dict({
            "obs": MultiDiscrete(np.full(len(pos_list)+1, 3), dtype=
                int),
            ENV_STATE: MultiDiscrete(np.full(len(pos_list), 3), dtype
                =int)
        }),
        Dict({
            "obs": MultiDiscrete(np.full(len(pos_list)+1, 3), dtype=
                int),
            ENV_STATE: MultiDiscrete(np.full(len(pos_list), 3), dtype
                =int)
        }),
    ])
```

```
    act_space = Tuple([
        PCBEnvironment.action_space,
        PCBEnvironment.action_space,
    ])
    register_env(
        "grouped_PCB",
        lambda config: PCBWithGroupedAgents(config).with_agent_groups
            (
             grouping, obs_space=obs_space, act_space=act_space))

    single_obs_space = Dict({
        "obs": MultiDiscrete(np.full(len(pos_list)+1, 3), dtype=int),
        ENV_STATE: MultiDiscrete(np.full(len(pos_list), 3), dtype=int
            )
    })

    # Now define the observation space for each agent, if they are
       different adjust accordingly
    obs_space_dict = {
        "agent_1": single_obs_space,
        "agent_2": single_obs_space,
    }
    action_space_dict = {
        "agent_1": PCBEnvironment.action_space,
        "agent_2": PCBEnvironment.action_space,
    }
```

6.1.2 Configuration for Qmix

```
    # Configure the QMix algorithm using the object-oriented approach
    #Configuration of the appoach Min varables to change
    qmix_config = QMixConfig()
    qmix_config.training(
        mixer="qmix",
        mixing_embed_dim=32,
        double_q=True,
        optim_alpha=0.99,
        optim_eps=0.00001,
        grad_clip=10.0,
        lr=0.0005,
        train_batch_size=32,
        target_network_update_freq=5,
        num_steps_sampled_before_learning_starts=10
    ).resources(
        num_gpus=int(os.environ.get("RLLIB_NUM_GPUS", "0")),
        # num_workers=1
```

```
    ).rollouts(
        rollout_fragment_length=4,
        batch_mode="complete_episodes"
    )
# Since the exploration method call did not work, configure
   exploration settings directly
    config_dict = qmix_config.to_dict()
    config_dict['exploration_config'] = {
        "type": "EpsilonGreedy",
        "initial_epsilon": 1.0,
        "final_epsilon": 0.1,
        "epsilon_timesteps": 40
    }

    config_dict.update({
        "env": "grouped_PCB",
        "env_config": {
            "separate_state_space": True,
            "one_hot_state_encoding": True
        },
        "multiagent": {
        "policies": {
                "pol1": (None, Tuple([ single_obs_space,
                   single_obs_space]),
                        Tuple([PCBEnvironment.action_space,
                           PCBEnvironment.action_space]), {
                    "agent_id": 0,
                }),
                "pol2": (None, Tuple([single_obs_space,
                   single_obs_space]),
                        Tuple([PCBEnvironment.action_space,
                           PCBEnvironment.action_space]), {
                    "agent_id": 1,
                }),
            },
            "policy_mapping_fn": lambda agent_id, *args, **kwargs: "
               pol1" if agent_id == "agent_1" else "pol2",
            "observations": obs_space_dict
        }
    })

    group = True
    start_time = time.time()
    ray.init(num_cpus=args.num_cpus or None)
    stop = {
        "episode_reward_mean": args.stop_reward,
        "timesteps_total": args.stop_timesteps,
    }
```

```
    config = dict(config_dict, **{
        "env": "grouped_PCB" if group else PCBEnvironment,
    })
################ training iteration##############
    results = tune.run(
        "QMIX",
        stop={"training_iteration": 150},
        config=config_dict,
        storage_path=trial_dir,
        callbacks=[MyCallback()],
        verbose=3,
        sync_config=SyncConfig(sync_timeout=3600)
    )
    ray.shutdown()
%###################
```

6.1.3 Reward Function

```
def cal_reward(state,agent):
    ################ read each Csv file
       ###################################
    global num_t, num_t1
    # with open('/content/sample1'+ str(file_number+1) +'.csv',
       newline='') as csvfile:
    used_decap = sum(state)
    if agent == 0:
        with open(user_name + r"\Documents\PI_MARL\H-board\Qmix\
           Design1.emc\PI-1/Power_GND/1-PIPinZ_IC1.csv", newline='')
           as csvfile:
            reader = csv.reader(csvfile, delimiter=';')
            next(reader, None)
            current_impedance_list = list(reader)
        frequency, impedance_values, num_rows1=fh.
           get_freqency_and_impedance(current_impedance_list)
        ########################## count ##############################
        count = 0  # how many points satisfy the target impedance
        for i in range(num_rows1):
            if impedance_values[i] <= fh.target_impedance(frequency[i
               ], fmin, fmax, fmittel, Zmin, Zmax):
                # datatype
                count += 1

        num_t = num_t1
        num_t1 = count
        empty_decap = state_einheit - used_decap
```

```
        if num_t1 == num_rows1:
            return ((num_t1 - num_t) / (num_rows1)) + (10 * (
               empty_decap / state_einheit)), True
        else:
            return (num_t1 - num_t) / (num_rows1), False
    if agent == 1:
        with open(user_name + r"\Documents\PI_MARL\H-board\Qmix\
           Design1.emc\PI-1/Power_GND/1-PIPinZ_IC2.csv", newline='')
           as csvfile:
            reader = csv.reader(csvfile, delimiter=';')
            next(reader, None)
            current_impedance_list = list(reader)
        frequency, impedance_values, num_rows2=fh.
           get_freqency_and_impedance(current_impedance_list)
        ######################### count ##############################
        count_2 = 0  # how many points satisfy the target impedance
        for i in range(num_rows2):
            if impedance_values[i] <= fh.target_impedance(frequency[i
               ], fmin, fmax, fmittel, Zmin, Zmax):
                # datatype
                count_2 += 1
        num_t = num_t1
        num_t1 = count_2
        empty_decap = state_einheit - used_decap
        if num_t1 == num_rows2:
            return ((num_t1 - num_t) / (num_rows2)) + (10 * (
               empty_decap / state_einheit)), True
        else:
            return (num_t1 - num_t) / (num_rows2), False
```

6.1.4 Single Agent Environment Code

```
class PCBEnvironment(MultiAgentEnv):

    action_list=fh.possible_action(pos_list, type_list)
    action_space =  Discrete(len(action_list))
    observation_space =  MultiDiscrete(np.full(len(pos_list),3),dtype
       =int)
    def __init__(self, env_config):
        super().__init__()
        self.action_pos = pos_list
        self.action_type = type_list
        self.action_list=fh.possible_action(pos_list, type_list)
        self.action_space =  Discrete(len(self.action_list))
        self.state = None
        self.C_ou = 0
```

```
        self.agent_1 = 0
        self.agent_2 = 1
        self._skip_env_checking = True
        self.preobs = np.zeros((len(pos_list)),dtype=int)
        self.actions_are_logits = env_config.get("actions_are_logits"
           , False)
        self.one_hot_state_encoding = env_config.get("
           one_hot_state_encoding", False)
        self.with_state = env_config.get("separate_state_space",
           False)
        self._agent_ids = {0, 1}
        if not self.one_hot_state_encoding:
            self.observation_space =  MultiDiscrete(np.full(len(
               pos_list),3),dtype=int)
            self.with_state = False
        else:
            if self.with_state:
                self.observation_space = Dict(
                    {
                        "obs": MultiDiscrete(np.full(len(pos_list),3)
                           ,dtype=int),
                        ENV_STATE: MultiDiscrete(np.full(len(pos_list
                           ),3),dtype=int)
                    }
                )
            else:
                self.observation_space =  MultiDiscrete(np.full(len(
                   pos_list),3),dtype=int)

        # every new episode
    def reset(self,seed=None, options=None):
        if seed is not None:
            np.random.seed(seed)
        global num_t, num_t1
        num_t = 0
        num_t1 = 0
        self.count = 0
        self.state= np.zeros((len(pos_list)),dtype=int)
        reset_pebfiles()
        return self._obs(), {}

    def step(self, action_dict):
        # initialize the current state after the selection of action
        actiona = [0 , 1]
        action = []
        action_list=fh.possible_action(pos_list, type_list)
        for i in range(2):
            actiona[i] = action_dict[i]
```

```
            action.append(action_list[actiona[i]])
            gen_pebfiles(action[i])

        next_obs = fh.next_state_step(self.state, action[0])
        self.state = next_obs  # Be cautious about shared state
           across agents
        next_obs = fh.next_state_step(self.state, action[1])
        self.state = next_obs  # Be cautious about shared state
           across agents
        # self.current_obs = self.state  # This might also need to be
            specific per agent
        self.preobs = next_obs
        # Handle file operations related to the action


        # Compute reward, check if the episode is done, etc.

        fh.load_batch_file(destination_file)
        Reward_value1, done1 = cal_reward(next_obs,0)
        Reward_value2, done2 = cal_reward(next_obs,1)

        if done1:
            info_terminated = {"message": "Environment reached
               terminal state"}
        if (self.count == state_einheit or self.count == len(pos_list
           )) and done1 == False:
            info_truncated = {"message": "Environment truncated
               before reaching terminal state"}
            Reward_value1 = -2
            Reward_value2 = -2
        rewards_dict = {self.agent_1: Reward_value1, self.agent_2:
           Reward_value2}
        self.count += 1
        obs = self._obs()
        terminateds = {self.agent_1: done1, self.agent_2: done2, "
           __all__": done1 and done2}
        truncateds = {"__all__": False}
        infos = {
            self.agent_1: {"done": terminateds[self.agent_1]},
            self.agent_2: {"done": terminateds[self.agent_2]},
        }
        self.C_ou += 1
        print(infos)
        if terminateds["__all__"]:
            # Convert complex objects to strings or flatten them as
               needed
            state_str = str(self.state)  # Example conversion, adjust
                as needed
```

```
            rewards_str = {k: str(v) for k, v in rewards_dict.items()
                } # Convert each value to string

            data_to_save = {
                'iteration': self.C_ou,
                'state': state_str,
                'rewards': rewards_str,
                'count': self.count
            }

            # Define the CSV file path
            csv_file_path = user_name + r"\Documents\PI_MARL\H-board\
                Qmix_termination_state_150_s.csv"

            # Write to CSV, appending each time
            with open(csv_file_path, 'a', newline='') as file:
                fieldnames = ['iteration', 'state', 'rewards', 'count
                    ']
                writer = csv.DictWriter(file, fieldnames=fieldnames)

                # Write headers if the file is newly created
                if file.tell() == 0:
                    writer.writeheader()

                writer.writerow(data_to_save)

            print("State appended to CSV file due to termination of
                both agents.")
        if sum(self.preobs)>state_einheit:
            reset_pebfiles()
        return obs, rewards_dict, terminateds, truncateds, infos

    def _obs(self):
        if self.with_state:

            obs_space = {
                self.agent_1: {"obs": self.agent_1_obs(), ENV_STATE:
                    self.state},
                self.agent_2: {"obs": self.agent_2_obs(), ENV_STATE:
                    self.state},
            }

            return obs_space
        else:
            return {self.agent_1: self.agent_1_obs(), self.agent_2:
                self.agent_2_obs()}
```

```
    def agent_1_obs(self):

        if self.one_hot_state_encoding:
            return np.concatenate([self.state, [1]])
        else:
            return np.flatnonzero(self.state)[0]

    def agent_2_obs(self):
        if self.one_hot_state_encoding:
            return np.concatenate([self.state, [2]])
        else:
            return np.flatnonzero(self.state)[0] + 3
```

6.1.5 Multi-agent Agent Environment Code

```
##############################################
class PCBWithGroupedAgents(MultiAgentEnv):
    def __init__(self, env_config):
        super().__init__()
        env = PCBEnvironment(env_config)
        tuple_obs_space = Tuple([env.observation_space, env.
           observation_space])
        tuple_act_space = Tuple([env.action_space, env.action_space])

        self.env = env.with_agent_groups(
            groups={"agents": [0, 1]},
            obs_space=tuple_obs_space,
            act_space=tuple_act_space,
        )
        self.observation_space = self.env.observation_space
        self.action_space = self.env.action_space
        self._agent_ids = {"agents"}
        self._skip_env_checking = True

    def reset(self, *, seed=None, options=None):
        return self.env.reset(seed=seed, options=options)

    def step(self, actions):
        return self.env.step(actions)
```