CHAIR OF ELECTRONIC DESIGN AUTOMATION

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Electrical Engineering and Information Technology

Microfluidic Design Automation based on Deep Reinforcement Learning in Parameterized Action Space

Shuo Wu

CHAIR OF ELECTRONIC DESIGN AUTOMATION

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Electrical Engineering and Information Technology

Microfluidic Design Automation based on Deep Reinforcement Learning in Parameterized Action Space

Mikrofluidik-Designautomatisierung basierend auf Deep Reinforcement Learning im parametrisierten Aktionsraum

Author:	Shuo Wu
Supervisor:	Prof. Dr.-Ing. Ulf Schlichtmann
Advisor:	Yushen Zhang
Submission Date:	2023.08.15

I confirm that this master's thesis in electrical engineering and information technology is my own work and I have documented all sources and material used.

Munich, 2023.08.15 Shuo Wu

Acknowledgments

I would like to express my sincere gratitude to my advisor, Yushen Zhang, for his invaluable guidance and support throughout the course of this thesis. His expertise and insights greatly contributed to the development and execution of this research. I would also like to extend my appreciation to Prof. Dr.-Ing. Ulf Schlichtmann for his insightful suggestions and constructive feedback, which significantly enriched the quality of this study.

Abstract

This study introduces a novel approach to automate microfluidic chip design using Deep Reinforcement Learning (DRL) in parameterized action space. This framework combines the DRL algorithm with microfluidic chip design strategy to optimize layouts for diverse objectives in the design progress of microfluidics. Key contributions of this work include the integration of DRL into design automation, addressing data limitations, and offering flexible chip design optimization.

A thorough review of existing literature reveals a gap in applying Deep Reinforcement Learning (DRL) to the comprehensive design of microfluidic chip layouts. Our proposed algorithm addresses this gap by abstracting the chip environment and utilizing a hybrid action space along with a customized reward system to make well-informed decisions. To enhance the training process, we employ various convergence strategies, resulting in efficient and effective chip designs.

Through experiments, our algorithm demonstrates its advantages in optimizing chip size, connection length, and computational efficiency. By comparing our approach to manual design and considering different convergence strategies, we outline both its strengths and limitations. Particularly, our algorithm stands out in chip size optimization and quick convergence, presenting promising real-world applications.

Contents

1 Introduction

Microfluidics has emerged as a fascinating interdisciplinary field that involves molecular analysis, biodefence, molecular biology, and microelectronics to manipulate and control fluids at the micrometer scale [1]. This technology has found diverse applications, ranging from chemical analysis to medical diagnostics, offering the potential for significant advancements in various industries. As the demand for sophisticated and efficient microfluidic devices (often in the form of chips) grows, the process of designing such devices becomes increasingly complex.

The intricate nature of microfluidic systems, combined with their ever-expanding range of applications, presents unique challenges in the design and optimization process. Engineers and researchers are tasked with creating microfluidic layouts that optimize fluid flow, minimize undesirable interactions, and enhance overall system performance. Achieving these goals requires a deep understanding of fluid dynamics, material properties, and precise control over device components. On the other hand, there are some processes of microfluidic chip design, such as the minimization of the chip size or avoiding the overlaps of microfluidic components or connections, that don't involve professional biology or chemistry knowledge. The optimization of these features can significantly influence the performance of the microfluidic chip, but the process can be time-consuming and may require the designer to learn additional knowledge about relevant optimization strategies in the field of general design automation.

In recent years, computational methods and tools have become indispensable allies in tackling the complexities of microfluidic chip design. Computer-aided design (CAD), simulation software, and optimization algorithms offer powerful means to explore a vast design space, predict fluid behavior, and iteratively refine device configurations. This synergy of traditional engineering principles and cutting-edge computational techniques has opened doors to innovative microfluidic solutions that were once challenging to envision.

This thesis explores the domain of planar microfluidic design automation, particularly emphasizing the use of reinforcement learning (RL) algorithms to improve the design process. RL has shown success in diverse areas, including games and robotics [2]. By using RL techniques in microfluidic design automation, the goal is to utilize their capabilities for creating intelligent and effective design approaches for microfluidic chips.

1.1 Motivation and Goal

The motivation behind the design automation of microfluidics is to enable chemists and biologists to focus on their expertise in component selection and specifications, rather than grappling with chip design intricacies. The complexity and time required for chip design, especially with numerous components or layers, can be a burden. Utilizing deep reinforcement learning, which combines reinforcement learning and deep learning, shows the potential to automate decision-making in the design process. This prompts the question: Can we train a model to make design decisions automatically?

The primary objective of this research is to create a link in an end-to-end microfluidic design and fabrication pipeline. Users input component specifications, and deep reinforcement learning algorithms handle decision-making and action formulation. The result is a detailed chip design in JSON. This design blueprint in JSON can be transformed into an STL format for 3D printing [3]. The process empowers chemists and biologists to translate their ideas into reality, while our system handles the complex tasks of microfluidic chip creation.

1.2 Brief Overview of the Approach

In this study, we present a comprehensive methodology that fuses microfluidic chip design principles with deep reinforcement learning algorithms. The key components of our approach include defining the microfluidic chip design space, formulating reward functions that quantify design performance, and employing an advanced RL algorithm called Deep Deterministic Policy Gradients (DDPG) modified with a parameterized action space.

The following chapters of this thesis explore our proposed approach in detail. We discuss the hierarchical policy network, the DDPG algorithm, and the techniques used to improve its performance. We also describe the experimental setup, share results, and address implementation challenges.

2 Literature Review

2.1 Microfluidic Design

Microfluidics is a rapidly evolving field that has attracted considerable attention due to its wide-ranging applications and potential to revolutionize various industries. This section provides an overview of key advancements in microfluidic design and simulations, highlighting the contributions and insights from existing research.

2.1.1 Microfluidic Design Approaches

Microfluidic design encompasses a diverse range of techniques and methodologies aimed at creating functional and optimized microscale devices. Early approaches often relied on manual design and fabrication processes, leading to time-consuming iterations and limited design exploration. With the advent of computer-aided design (CAD) software, researchers gained the ability to create detailed layouts and prototypes digitally, significantly accelerating the design process. Elishai has done an overview of CAD applied in the field of microfluidic design in 2020 [4]. The review provides insights into recent advancements in the realm of computer-aided design for various types of microfluidic systems, including flow-based, droplet-based, and paper-based microfluidics. A specific case study about the design of resistive microfluidic networks is explored in-depth, showcasing the application of these design approaches. The article highlights the evolution of microfluidic devices, characterized by increased complexity, enhanced performance requirements, novel materials, and innovative fabrication techniques. As a result, the field has witnessed the emergence of new algorithms and design methodologies aimed at optimizing and facilitating the entire design process of microfluidic circuits, spanning from conceptualization and specification to synthesis, realization, and refinement. This progress encompasses the development of fresh description languages, optimization techniques, benchmarking approaches, and integrated design tools.

2.1.2 Microfluidic Design Description

Microfluidic designs are typically formulated using dedicated editors that allow the incorporation of pre-defined microfluidic components from libraries. A critical advance-

ment in Computer-Aided Design (CAD) for microfluidics pertains to the development of hardware description languages. These languages facilitate the representation of microfluidic components as code, offering a consistent and abstract layer that decouples software development from potential technological changes, as in the work of Thies's microfluidic layer abstraction [5]. Huang also proposed a utilized description language, MINT, based on modules and was highly utilized, as a preparation for his end-to-end workflow of microfluidic design called Fluigi [6].

2.1.3 Design for Optimizationn

CAD methodologies cater to both general and specialized microfluidic applications, warranting the optimization of design layouts to meet specific objectives. The optimization process may entail determining the ideal arrangement of components, sequences of operations, and allocation of resources. The synthesis of optimization parameters and component weighting establishes a loss function for effective design optimization.

Microfluidic design optimization employs iterative, heuristic-based, and exact analytical methods. Iterative approaches iteratively refine a base solution to achieve convergence. Heuristic-based techniques provide rapid but approximate solutions, while analytical methods strive to yield the optimal solution either by exhaustive exploration or mathematical derivation. Design for X principles proposed by Gatenby extend optimization to factors such as testability, assembly, and serviceability, particularly important in addressing fabrication challenges [7].

For example, Tseng et al. proposed an optimization of a balance between device size and execution time by altering the layout of planar microfluidics and the number of components such as mixers [8]. Many other optimization works have been done focusing on different problems involving microfluidic chip design. They've also presented a valve-centric optimization approach, wherein designs are systematically optimized to minimize the frequency of valve-switching operations, thereby contributing to the extension of a device's operational longevity. Hu has proposed an optimization task that minimizes the control-pin and response time by improving routing in control-layer [9]. As an illustration, Lin et al. introduced an algorithm aimed at minimizing the cumulative length of flow channels [10]. More recently, Yang et al. extended prior research by relaxing fundamental assumptions that constrained routing to adhere strictly to Manhattan routing metrics, characterized by straight channels and 90° bend [11]. This relaxation enabled the routing of channels at arbitrary angles, resulting in a notable reduction of channel length by over 15%.

Instead of focusing problems within one layer, there are also many attempts to optimize the synthesis of microfluidic chips overall. As an illustration, Wang et al. introduced a placement algorithm that optimizes the arrangement of fluidic com-

ponents by minimizing congested regions within channels [12]. A similar strategy named Columba was introduced by Tseng et al., which extended Wang's approach by incorporating angled channel routing and pressure-sharing control channels [13]. Recognizing the prevalent integration of existing modules in microfluidic design, Li et al. proposed the concept of component-oriented synthesis. This paradigm treats conventional microfluidic systems as discrete entities, categorized into either containers, comprising chambers formed by valve-bound channel segments with circulating flows, or accessories, encompassing pumps, heating pads, optical sensing units, specialized sieve valves impeding particle flow, and cell traps. Within this framework, operations are defined by a combination of containers and accessories, along with their temporal durations and interdependencies [14]. The resulting sequence graph represents operations and their relationships, which necessitates scheduling on the microfluidic device.

2.1.4 Benchmarking Microfluidic Designs

The multifaceted nature of microfluidic applications necessitates the establishment of benchmarks to compare diverse design solutions. A microfluidic benchmark consists of curated designs generated for specific applications using varied optimization methods. These benchmarks enable objective comparisons based on established metrics, offering insights into the suitability of designs across different contexts.

Crites et al. introduced a standardized interchange format referred to as ParchMint for continuous microfluidics [15]. This format employs a JavaScript Object Notation (JSON) file to specify the device netlist, consisting of a list of connections. The ParchMint standard delineates the device's architectural elements, including components, connections, and layers, which collectively define the device's intricate structure. ParchMint's utility extended to the creation of a microfluidic benchmark, incorporating reverse-engineered designs derived from images, generic layouts of cell traps and valves, as well as other designs derived from specific applications. These designs were translated into a benchmark space, characterized by the number of components and dimensions, enabling researchers to compare their designs against counterparts of similar complexity.

2.1.5 Integration of Design Tools

The integration of microfluidic description languages, optimization algorithms, and benchmarks within CAD frameworks is a significant challenge due to differing development contexts. A key instrument in this realm is the integrated development

environment (IDE), which empowers designers to define, optimize, and prepare microfluidic device designs for fabrication.

Tseng et al. introduced the Columba design synthesis tool, a co-layer optimization solution as previously discussed. This tool takes device specifications and transforms them into an optimized microfluidic layout, subsequently translating it into a sequence of AutoCAD drawing commands. By facilitating a seamless transition from specification to device fabrication, Columba streamlines the design process. Notably, a comprehensive and extensively employed design tool is 3DμF, a creation of Sanka et al. [16]. This open-source and interactive microfluidic design tool establishes an efficacious visual platform to serve future design automation algorithms, fabrication techniques, and control mechanisms. The framework adopts ParchMint for layout representation, furnishes a repository of parameterized microfluidic blocks, and supports modular microfluidic configurations.

2.2 Reinforcement Learning

Reinforcement Learning (RL) algorithms have garnered significant attention in the field of artificial intelligence and machine learning due to their remarkable ability to enable agents to learn optimal strategies for sequential decision-making tasks through interaction with an environment. In recent years, RL has found application in diverse fields such as robotics, finance, healthcare, recommendation systems, and game playing. Ongoing research focuses on improving sample efficiency, stability, and addressing challenges like exploration in high-dimensional continuous spaces. Additionally, the integration of RL with other techniques, such as imitation learning and meta-learning, continues to push the boundaries of reinforcement learning's capabilities. This section aims to provide an overview of the various RL algorithms, highlighting their key characteristics, advancements, and applications as well as to provide a theoretical basis for the subsequent task selection algorithm

2.2.1 Deep Q-Networks (DQN)

In the context of Q-learning or DQN, an import concept is ***Q-value***, which represents the expected cumulative reward that an agent can obtain by taking a particular action in a specific state and the following rewards when taking actions generated by a certain policy thereafter. DQN extends Q-learning by utilizing deep neural networks to approximate these Q-values. This advancement enables DQN to handle high-dimensional state spaces, making it suitable for tasks like image-based game-playing. Notable improvements include experience replay and target networks to stabilize training and improve convergence [17].

2.2.2 Policy Gradient Methods

Policy gradient methods directly optimize policy functions to maximize expected cumulative rewards. They employ gradient ascent on objective functions to update policies incrementally. Algorithms like REINFORCE and Trust Region Policy Optimization (TRPO) fall under this category [18]. Policy gradient methods are well-suited for continuous action spaces and have been successfully applied in robotic control and autonomous systems.

2.2.3 Actor-Critic Methods

Actor-critic methods combine elements of both value-based and policy-based approaches [19]. They maintain two networks: the actor, which generates actions, and the critic, which estimates the value of states. Advantage Actor-Critic (A2C) and Asynchronous Advantage Actor-Critic (A3C) are popular variants. These algorithms achieve a balance between exploration and exploitation, enabling faster convergence. A2C is an algorithm that aims to address some of the limitations of traditional policy gradient methods by incorporating the idea of value estimation into the learning process. The A2C algorithm uses a combination of policy gradient updates based on the advantage function and value function updates to improve both the agent's policy and the accuracy of its value estimation. By training these two components simultaneously, A2C tends to learn more efficiently and converge faster than pure policy gradient methods. A3C builds upon the A2C algorithm but introduces parallelism to accelerate training by using multiple agents that interact with their instances of the environment simultaneously. The asynchronous nature of A3C reduces the correlation between consecutive updates and introduces exploration diversity, which can lead to more stable learning and faster convergence. However, managing asynchronous updates and ensuring proper exploration can be challenging due to potential race conditions and other synchronization issues [20].

2.2.4 Proximal Policy Optimization (PPO)

PPO is an on-policy policy optimization algorithm that aims to enhance stability and sample efficiency [21]. It employs clipped surrogate objectives to prevent policy updates from deviating too far. PPO has gained prominence for its ease of use, robustness, and successful application in various domains.

2.2.5 Deep Deterministic Policy Gradient (DDPG)

DDPG is designed for continuous action spaces and combines DQN's value function approximation with policy gradient methods [22]. It employs actor and critic networks to learn deterministic policies and value functions. DDPG is widely used in robotic control and autonomous systems due to its ability to handle continuous control tasks.

2.2.6 Reinforcement Learning with parametrized action spaces

As the design automation of microfluidic chips is a complex procedure that involves both discrete actions such as selecting a component and continuous actions such as moving or rotating a component, a parametrized action space is unavoidable in our scenario. This section focuses on some reinforcement learning algorithms that incorporate parametrized action space. These algorithms demonstrate how parametrized action spaces enhance the capabilities of reinforcement learning methods, enabling agents to tackle complex tasks with precise and adaptable control strategies. Parametrized actions provide flexibility and versatility, making them well-suited for various real-world applications in robotics, control systems, and autonomous agents.

PADDPG (Parametrized Action Deep Deterministic Policy Gradient)

PADDPG is a popular algorithm for continuous action spaces with parametrized actions [23]. It employs an actor-critic architecture, where the actor-network learns a deterministic policy with parametrized actions, and the critic network estimates the value function. PADDPG combines value-based and policy-based methods, enabling it to handle parametrized action spaces efficiently. This makes it well-suited for tasks that require precise and flexible control strategies, such as robotic manipulation and complex autonomous systems. The incorporation of parametrized actions in PADDPG allows agents to explore a wider range of actions while maintaining smooth and adaptable policies. As a result, PADDPG has shown promising results in various applications, making it a versatile choice for reinforcement learning in scenarios with complex and continuous action spaces.

2.3 Programming language and framework

Choosing a programming language and framework is a critical decision before jumping into the concrete implementation. For reinforcement learning projects, there are some common choices for programming languages, such as Python, C++, and Julia. As one of the most widely used programming languages, Python is a high-level, general-purpose

programming language with a rich set of machine learning and reinforcement learning libraries, such as TensorFlow, PyTorch, and OpenAI Gym [24, 25, 26]. Tensorflow and Pytorch are both very popular choice of framework in the field of deep learning. PyTorch is a Python-based open-source machine learning framework built upon the Torch library. PyTorch uses a dynamic computational graph, making it intuitive for research and experimentation, allowing real-time debugging and interactive prototyping [27]. Tensorflow, on the other hand, also has its own advantages as it has a more mature ecosystem and better deployment options [28]. After comparison, the final decision was to utilize the combination of Python and PyTorch to implement this work, as demonstrated in the upcoming code segments. The goal is to provide a perspective and example of algorithm implementation. The programming language and framework can be adjusted based on user requirements.

3 Methodology

3.1 Task Desciption

This thesis is fundamentally focused on design automation which is a crucial link in the end-to-end microfluidic design and fabrication process. Initially, users input their chosen components with specific details like size, interface positions, and connections. With the help of well-trained deep reinforcement learning networks, the algorithms take charge of decision-making and action planning in the design phase. This results in a carefully crafted chip design plan that includes component placement and connections, presented in a structured format like JSON. This format remains intelligible within microfluidic design environments, such as Flui3D [3], capable of transforming our blueprint into an STL format that can be subsequently employed by a 3D printer for microfluidic fabrication. This comprehensive process quickly transforms the creative concepts of chemists and biologists into tangible outcomes. By entrusting our system with the complexities of design and fabrication, these experts can focus solely on tackling challenges in the fields of chemistry and biology.

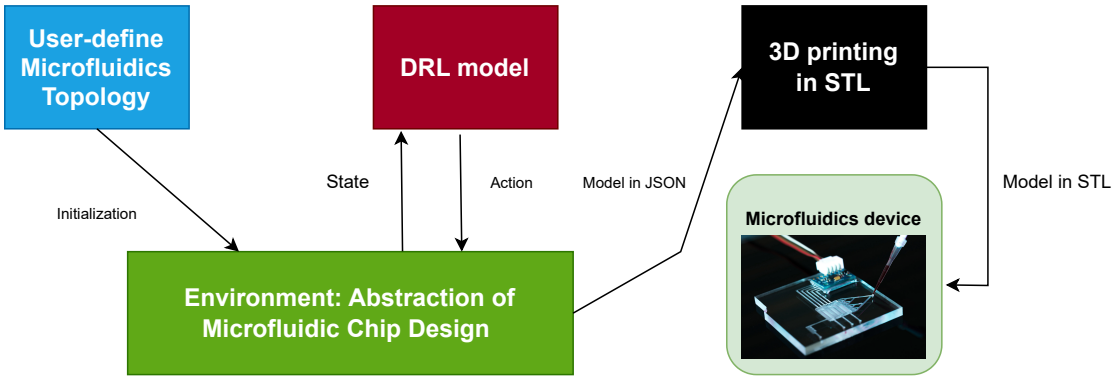

Figure 3.1: DRL model in the end-to-end microfluidic design and fabrication chain.

3.2 Abstraction of Microfluidic Chip Design

In microfluidic chip design, essential concepts provide the foundation for constructing complex fluidic systems. This section introduces the approach to simplify, simulate, and specify a microfluidic chip design. This simplified design will act as the environment for interaction with the deep reinforcement learning algorithm in the following section. Therefore, the developed model should encompass additional functionalities such as environment reset, state change after actions, and reward calculation..

3.2.1 Component Representation: *Comp* Class

Microfluidic chip components serve as the fundamental building blocks, shaping the intricate pathways of fluidic movement. The ***Comp*** class acts as a versatile abstraction, encompassing essential attributes including layers, position, orientation, and pin configuration. This class not only facilitates the calculation of vertices based on these properties but also extends its utility by representing various component types, such as mixers, separators, and functional entities, each manifesting distinct roles within the chip's design.

```
class Comp:
    def __init__(self, comp_id, width, height, x, y, th, layer, n_in, n_out):
        # Initialize component attributes
        self.id = comp_id
        self.width = width
        self.height = height
        self.x = x
        self.y = y
        self.th = th
        self.layer = layer
        ...

    def calculate_vertices(self):
        # Calculate component vertices
        ...
```

In the field of microfluidic chip design, a variety of components come into play, each playing a role in the chip's overall functionality. These components, such as mixers and separators, perform distinct tasks essential for fluid manipulation. By abstracting these components into rectangular entities with defined pins, the ***Comp*** class offers a unified framework for modeling a variety of microfluidic chip elements. 3.2 from

Flui3D [3] shows some common types of microfluidic which can be easily represented by a rectangular contour with ports to connect with other components.

It's worth noting that during the abstraction process, some components such as droplets and width transitions in 3.2, when represented as rectangles, may result in the wastage of potential wiring space, thereby leading to a certain degree of chip area inefficiency. However, irregular shapes introduce a significant computational burden during subsequent reward calculations. This computational overhead makes it more challenging for the neural network being trained to converge effectively. Moreover, due to the diverse array of shapes that components can be abstracted into, along with their dependence on numerous parameters, the introduction of complex reward calculations is a concern. To ensure model generalizability and avoid overly intricate reward computations, it was decided to abstract all components as rectangles during the abstraction process. This approach involves distinguishing components based on variations in length, width, and the distribution of ports, striking a balance between computational efficiency and the model's ability to accommodate a wide range of scenarios.

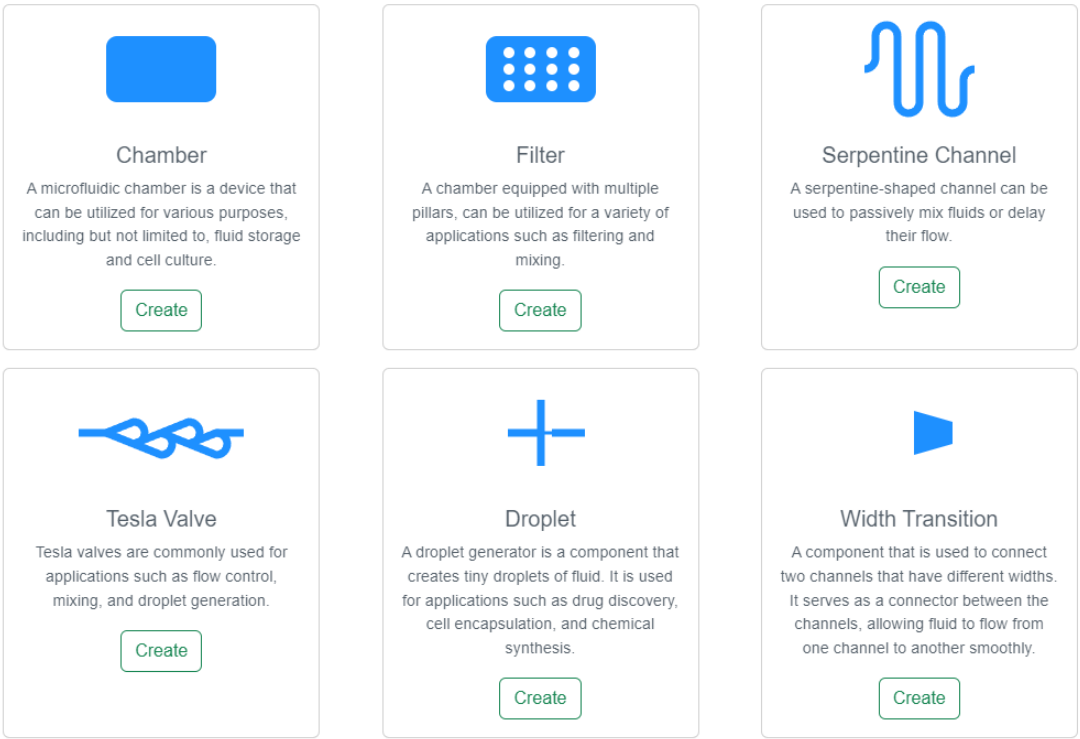

Figure 3.2: Common types of microfluidic components from Flui3D [3].

3.2.2 Port Representation: *Port* Class

Ports, sometimes referred to as pins, play a crucial role in microfluidic chip design by facilitating fluid connections between components. The ***Port*** class encompasses key attributes that define the unique properties of a port. These attributes include the associated component's identifier, the port's unique identifier, its relative position within the component along the x-axis (width), its relative position along the y-axis (height), and its directionality.

```
class Port:
    def __init__(self, comp_id, port_id, rx, ry, direction):
        self.comp_id = comp_id
        self.port_id = port_id
        self.rx = rx
        # relative to the port position on the component's x-axis (width)
        self.ry = ry
        # relative to the port position on the component's y-axis (height)
        self.direction = direction
        self.x = 0
        self.y = 0
```

As per the definition of ports, each port is exclusively positioned along one of the four edges of a component rectangle. Therefore, the coordinates of a port are represented relative to the corresponding component. Upon analyzing various types of components, it becomes apparent that a considerable portion of them exhibit port distributions characterized by axial symmetry or by being situated solely along two non-adjacent edges of the component rectangle. In the subsequent ***reset*** method of the environment, the likelihood of symmetric port distributions will be augmented rather than being purely randomized. This augmentation aims to better emulate the probability of encountering different chip layouts during actual chip design. This approach aligns with the intent of achieving a more realistic simulation of the diverse probabilities associated with chip designs of varying complexities.

3.2.3 Interface Representation: *FreePort* Class

In microfluidic chip design, the concept of connecting the chip with external systems is of great importance. This is where the ***FreePort*** class becomes relevant - an abstraction that symbolizes the link through which the chip connects with external elements. Unlike conventional ports, the ***FreePort*** introduces a distinct component type, manifesting as a circular entity defined by a specified radius, as opposed to regular ports, which are represented as dots on the edges of the component rectangular. This unique

geometric configuration imparts the ***FreePort*** with its distinctive attributes, allowing it to effectively serve as an interface element.

```
class FreePort:
    def __init__(self, port_id):
        self.port_id = port_id
        self.x = 0 # Initial x coordinate
        self.y = 0 # Initial y coordinate
```

Functioning as a unique component, a ***FreePort*** allows two-way interaction. It can be visualized as a circular entity with a set radius. Its positioning creates a distinct link between the chip and external systems. Unlike regular ports that have only one connection point, a ***FreePort*** can manage several connections while staying at its central location. This key trait sets it apart, making it act both as a connector and a component for further actions and calculations.

3.2.4 Connection and Node Representation: *Connection* and *Node* Classes

In microfluidic chip design, fluid movements rely on connections and nodes working together. The ***Connection*** class is a key element that helps to make pathways for fluids by connecting ports and components. Each connection has a unique ID and links starting and ending ports, as well as a specific layer in the chip's structure.

```
class Connection:
    def __init__(self, conn_id, start_port_id, end_port_id, layer):
        self.conn_id = conn_id
        self.start_port_id = start_port_id
        self.end_port_id = end_port_id
        self.layer = layer
        self.nodes = []

    def add_node(self, node_id, x, y):
        node = Node(self.conn_id, node_id, x, y)
        self.nodes.append(node)

class Node:
    def __init__(self, conn_id, node_id, x, y):
        self.conn_id = conn_id
        self.node_id = node_id
        self.x = x
        self.y = y
```

Each connection, in practice, could take the form of a flexible curve with fixed start and end points. However, in the actual design process, designers often introduce various constraints to simplify the problem, such as in [13, 29]. In this work, each connection is defined as a polyline with a limited number of vertices, where these vertices correspond to instances of the Node class. The maximum number of vertices directly impacts the dimensionality of the state generated by the environment and the complexity of the optimization problem. After numerous attempts, it has been deduced that the optimal range for the maximum number of nodes per connection should be set between 3 and 10. Too few nodes might prevent the connection from maneuvering around components, while an excessive number of nodes could impede model convergence.

3.2.5 Microfluidic Environment: *ChipBoardEnv* Class

Microfluidic chip design centers on managing a flexible environment that includes components, ports, connections, and nodes. The ***ChipBoardEnv*** class represents this core concept and acts as a simulation area where chip layouts are created, studied, and improved. This class includes various features and actions that define the microfluidic world and aid in automated design.

```
class ChipBoardEnv:
    def __init__(self, max_n_comp, min_n_comp,
        max_ports_per_comp, max_n_connect,
        max_nodes_per_connect, max_n_free_port...):
        # Initialize ChipBoardEnv attributes
        ...

    def reset(self):
        # Reset the microfluidic environment to its initial state
        ...

    def step(self, action):
        # Perform a step in the environment based on the given action
        ...

    def render(self):
        # Visualize the current state of the microfluidic layout
        ...

    def calculate_reward(self):
        # Calculate the reward based on the current state
```

```
    ...

def generate_random_component(self):
    # Generate a random component with specified attributes
    ...

def generate_random_connection(self):
    # Generate a random connection between ports
    ...

def generate_random_freeport(self):
    # Generate a random free port
    ...
```

The ***ChipBoardEnv*** class captures how microfluidic chip design works. Its settings describe the environment, like the maximum number of components, ports, and connections allowed, as well as the size and poses of each component. The ***reset*** method starts the environment fresh for a new design, providing the reinforcement learning algorithm with a randomized initial state and combinations of different components and connections. The step method moves the environment forward based on an action, providing a new state as feedback to the reinforcement learning agent. The abstracted microfluidic layout can be easily illustrated using the visualization method, giving an intuitive description of the chip's structure. The environment's reaction to actions is measured with the calculate_reward method, supporting the learning process for better chip designs.

Inside the ***ChipBoardEnv*** class, multiple supporting methods make it easier to create, handle, and adjust components, ports, and connections. This versatile toolkit gives designers the ability to try out, examine, and improve microfluidic chip designs, and also prepares for subsequent neural network training.

3.2.6 Essential Methods of Microfluidic Environment

Reset of Microfluidic Environment

The ***reset*** method plays a crucial role in preparing the microfluidic environment for new design exploration. The workflow of this method is:

- The method clears the existing lists for components, ports, free ports, connections, and nodes, essentially resetting the environment to its initial state.

- The method manages the random seed to ensure reproducibility of experiments if required. The seed is either generated based on the system time or set to a saved value.
- The method generates a random number of components within the specified range (min_n_comp to max_n_comp) and creates instances of the Comp class for each component. Each component's attributes, such as width, height, and number of input and output ports, are determined randomly or based on specified probabilities.
- The number of free ports and inner connections is calculated based on the generated components, ensuring a balanced distribution of connections. Free ports are created and initialized.
- The method establishes initial connections between ports and free ports, taking into account the directions of the ports. It ensures a balanced distribution of connections to start the design process.
- If the number of free ports exceeds the maximum allowed, the environment is reset, preventing any inconsistencies.
- The ***reset*** method concludes by returning the initial state of the environment, which is crucial for starting the design exploration process.

The extensive procedure within the ***reset*** method establishes the foundation for experimenting with microfluidic chip design. It allows the generation of various layouts and setups, paving the way for subsequent optimization and exploration.

State space

The ***calculate_state*** method is responsible for generating the state representation of the current microfluidic environment. This representation will be used by the reinforcement learning agent to make informed decisions during the design process. The workflow of this method is:

- The method begins by initializing lists to store various components of the state representation, such as attributes of components, ports, free ports, connections, and nodes. The ***state_head*** list is populated with the lengths of different lists, providing context about the environment's current state.
- The lists are then padded to ensure they have the same length, as required for processing by the reinforcement learning algorithm. This is achieved by adding zero values to the lists if needed.

- The method proceeds to loop through components, ports, free ports, and connections, extracting their attributes and appending them to the corresponding state lists. For connections, information about associated nodes is also collected and added to the ***node_state*** list.
- After looping through all relevant components, ports, and connections, the ***node_state*** list is concatenated with the other state lists, and the entire representation is converted into a numpy array.
- The resulting state array encapsulates the current state of the microfluidic environment, including key attributes of components, ports, free ports, connections, and nodes. This state representation serves as input for the reinforcement learning algorithm, enabling the agent to learn and optimize microfluidic chip designs over successive iterations.

Step method and action space

The ***step*** method simulates one step of interaction between the agent and the environment. It takes an action as input, which consists of a high-level action and corresponding low-level action details. The method begins by extracting the high-level action and low-level action details from the input.

Based on the high-level action, the method dispatches the appropriate low-level action method to execute. The available low-level actions are placement, node insertion, node deletion, change component layer, and change connection layer. Each low-level action method modifies the state of the microfluidic environment based on the provided details.

After executing the low-level action, the method calculates the reward based on the updated state using the ***calculate_reward()*** method. The new state is also calculated using the ***calculate_state()*** method. If visualization is required, the method calls ***visualize_board()*** to display the updated microfluidic chip layout. Finally, the method returns the new state, reward, and a boolean value indicating whether the episode is done or not.

The ***step*** method is crucial for the agent's interaction with the environment and forms the core of the reinforcement learning training loop. It allows the agent to take actions, observe the outcomes, and learn to optimize microfluidic chip designs over time. The action space consists of five distinct types of actions that can be taken within the environment:

Placement Action: This action involves changing the position and orientation of a component or a free port. It is primarily used for moving components or free ports within the chip. The parameters required for this action are:

- ***placement_id***: The ID of the component or free port being moved. Positive values represent components, while negative values represent free ports.
- ***dx***: The displacement along the x-axis (horizontal movement).
- ***dy***: The displacement along the y-axis (vertical movement).
- ***dth***: The change in orientation (in radians).

Node Insert Action: This action involves inserting a new node into an existing connection, effectively adding a bend point to the connection. The parameters required for this action are:

- ***connection_id***: The ID of the connection in which the node is to be inserted.
- ***node_insert***: The index of the position in the connection's nodes list where the new node should be inserted.
- ***node_x***: The x-coordinate of the new node's position.
- ***node_y***: The y-coordinate of the new node's position.

Node Delete Action: This action involves deleting a node from an existing connection, effectively removing a bend point from the connection. The parameters required for this action are:

- ***connection_id***: The ID of the connection from which the node is to be deleted.
- ***node_delete***: The index of the node to be deleted within the connection's nodes list.

Change Component Layer Action: This action involves changing the layer (or plane) in which a component is placed. It allows components to be moved between different layers. The parameter required for this action is:

- ***comp_id***: The ID of the component whose layer is to be changed.

Change Connection Layer Action: Similar to the previous action, this one involves changing the layer of a connection. It allows connections to be moved between different layers. The parameter required for this action is:

- ***connection_id***: The ID of the connection whose layer is to be changed.

These actions together create the action space for the microfluidic chip design environment. Each action type comes with specific parameters that dictate how it works, and the mentioned methods above carry out these actions according to their parameters. In the environment, the step method carries out the selected action, updates the environment's state, and computes the reward based on the new state.

Reward method

The process of calculating rewards in the microfluidic chip design environment is thorough and encompasses the assessment of multiple facets of chip design to ascertain the value of a reward for an agent's action. These rewards are computed by considering a range of distinct components and factors.

```
def calculate_reward(self):
    overlap_punishment = self.calculate_overlap_punishment()
    chip_size = self.calculate_chip_size()
    conn_length = self.calculate_conn_length()
    n_vertical_tunnels = self.calculate_vertical_tunnels()
    comp_size = self.calculate_comp_size()
    n_conn = self.calculate_conn_complexity()

    # Dictionary for the reward components
    self.reward_dict = {
        "overlap": {"func": self.calculate_overlap_punishment, "weight": -100},
        "chip_size": {"func": self.calculate_chip_size, "weight": -2},
        "conn_length": {"func": self.calculate_conn_length, "weight": -1},
        "vertical_tunnels": {"func": self.calculate_vertical_tunnels, "weight": -1},
        "comp_size": {"func": self.calculate_comp_size, "weight": 3},
        "n_conn": {"func": self.calculate_conn_complexity, "weight": 1}
    }

    # Calculate the reward based on different components
    reward = 0
    for key, value in self.reward_dict.items():
        reward += value["weight"] * value["func"]()

    return reward
```

Here's a breakdown of the reward calculation process and the associated methods:

- ***overlap***: This component of the reward penalizes the overlapping of components

on the chip, which can lead to undesirable fluidic interactions.

- ***chip_size***: This component penalizes larger chip sizes, encouraging more compact designs.
- ***conn_length***: This component penalizes longer fluidic connections, promoting shorter and more efficient connections.
- ***vertical_tunnels***: This component penalizes the presence of vertical tunnels, which may complicate fabrication or increase the complexity of the design.
- ***comp_size***: This component rewards larger individual components, incentivizing the use of larger components when appropriate.
- ***n_conn***: This component rewards designs with higher complexity in their fluidic connections.

For each of the components mentioned above, a dictionary ***reward_dict*** is created, containing a reference to the corresponding calculation method ***(func)*** and a weight that determines the relative importance of that component in the final reward computation. In the context of microfluidic chip design, the issue of component or connection overlap is considerably more serious than a slightly larger chip size. Hence, the weights assigned to overlap and chip size should both be negative, with the weight magnitude of overlap distinctly greater. However, if there is a substantial disparity in the weights assigned to rewards or penalties (if they have negative weights), it can lead to significant gradient differences during the training of reinforcement learning, resulting in a challenging convergence of the model. The weights for individual rewards can be fine-tuned as hyperparameters, considering the normalizing rewards to mitigate this disparity is also an option.

In our work, the reward method iterates over the components in ***reward_dict***, calculating the weighted contribution of each component to the overall reward. The final reward is obtained by summing up the weighted contributions of all components.

The computed reward value serves as a quantitative measure of the quality of the microfluidic chip design. A higher reward value indicates better designs according to the specified metrics, while a lower value indicates less optimal designs. This reward value is crucial for reinforcement learning algorithms, as it guides the learning process by providing feedback on the desirability of different design decisions.

The first element of the dictionary is the **Overlap** meaning that the overlapping between the components and connections is penalized. This is split into three parts:

- ***component overlap (calculate_comp_overlap_punishment)***

- ***component-conn overlap (calculate_comp_conn_overlap_punishment)***
- ***connection overlap (calculate_conn_overlap_punishment)***

Take the calculation of overlap between components for example:

```
def calculate_comp_overlap_punishment(self):
    overlap_punishment = 0
    for i, comp1 in enumerate(self.Comps):
        for j, comp2 in enumerate(self.Comps):
            if i != j and comp1.layer != comp2.layer:
                rect1_coords = comp1.calculate_vertices()
                rect2_coords = comp2.calculate_vertices()
                poly1 = shapely.Polygon(rect1_coords)
                poly2 = shapely.Polygon(rect2_coords)
                intersection = poly1.intersection(poly2)
                if not intersection.is_empty :
                    overlap_punishment += intersection.area
    return overlap_punishment
```

The ***calculate_comp_overlap_punishment*** method calculates the degree of overlap between different components in a microfluidic chip design. It does this by iterating through pairs of components, checking if their bounding rectangles intersect, and adding up the area of overlap. This helps to identify and penalize overlapping regions between components, which can lead to problems in fluidic interactions or fabrication. The method returns the total overlap punishment value. The calculations of overlap between components and connections or overlap between connections only are similar to the calculation of overlap between components. However, it's worth noting that only components or connections within the same layer can count as overlap. The total overlap punishment is the sum of these three overlap penalties.

Next is the **Chip Size**. The size of the chipboard is calculated based on the position of components, ports, and nodes. The ***calculate_chip_size*** method computes the area of the chipboard with a safe distance added around the components and connections. **Connection Length** is also important to be taken into account when calculating the total reward. Without incorporating this reward, the trained model could excessively increase the number of nodes in the connections to minimize the overall chip area, potentially leading to a degradation in the chip's overall performance.

Another reward element worth mentioning is **the number of vertical tunnels**, where connections across different layers, are counted. The ***calculate_vertical_tunnels*** method determines the number of vertical tunnels within the chip. Changing layers of components and connections is a simple way of avoiding overlapping penalty, but also add

complexity to the chip design and fabrication. To avoid abuse of this action, using the number of vertical tunnels as a punishment, a reward with negative weight is necessary. This allows the trained agent to only change the layer of components or connections when it's necessary.

The **complexity of connections** is considered based on the number of connections, free ports, and components. The ***calculate_conn_complexity*** method computes a complexity score that takes into account the number of connections, free ports, and components. Each of these factors contributes to the overall reward. The ***calculate_reward*** method adds up the weighted values of these individual reward components to compute the final reward. The more complex the input microfluidic topology is, the more likely that the optimized layout has a lower reward since the reward elements involve overlapping penalty and chip size penalty. Therefore, it's reasonable to add the total size of components and complexity of connections as a positive reward to compensate for the unfair reward baseline of different input topologies.

The reward calculation process takes both positive and negative aspects of the chip design into account, allowing the agent to learn and optimize its actions to achieve a desired chip layout while avoiding undesirable configurations.

Visualization

The ***visualize_board*** method is responsible for creating a visual representation of the chipboard layout and updating it dynamically as the agent takes actions. This visualization provides a graphical representation of the chip design environment and allows users to monitor the progress of the agent.

Here's an explanation of the key components of the ***visualize_board*** method:

- Setting Up the Visualization: This part checks if the visualization thread is already running. If not, it starts a new figure for the visualization.
- Plotting Components, Ports, and Free Ports: This part plots the components' contours using their vertices. Ports and free ports are represented by red ***('ro')*** and green ***('go')*** circles, respectively. Then it initializes a new figure ***(plt.figure)*** with a specified size and title.
- Plotting Connections: This part plots connections using line segments based on the start and end ports. If the connection has nodes, it plots broken line segments connecting the nodes. Different colors are used to distinguish connections on different layers.
- Drawing Contour and Aspect Ratio: This part calls the ***calculate_chip_size***

method with draw=True to draw the chip's contour with a safe distance and sets the aspect ratio to "equal" to prevent distortion in the visualization.

- Adding Annotations: Adds text annotations to the plot to display various information, such as reward, action, episode number, etc.
- Updating the Visualization: Updates the annotations with the latest values (e.g., reward, action, episode number). Use ***plt.pause(0.1)*** to refresh the plot and make it interactive.
- Enabling Interactive Mode: Enables interactive mode using ***plt.ion()*** to allow continuous updating of the plot. The ***visualize_board*** method helps the user visualize the chipboard layout, the positions of components, ports, free ports, and connections, as well as the changes caused by the agent's actions.

This real-time visualization aids in understanding the chip design process and assessing the agent's performance. The effects of visualization are presented in the Figure 4.11, and it can be visualized animatedly to observe how the agent optimizes the layout through a sequence of actions during the training process.

3.3 Reinforcement Learning Algorithm

Reinforcement Learning (RL) is a machine learning paradigm focused on training agents to make optimal decisions within a specific environment. In RL, an agent interacts with an environment over time, taking actions to achieve a certain goal. The agent learns by receiving feedback in the form of rewards or penalties based on the actions it takes [30, 31].

Deep reinforcement learning (DRL), an amalgamation of reinforcement learning (RL) and deep learning paradigms, has emerged as a transformative approach capable of addressing an expansive spectrum of intricate decision-making challenges [32]. This innovative framework has transcended prior limitations, enabling the resolution of tasks that were hitherto deemed infeasible or yielded suboptimal outcomes. By synergistically harnessing the strengths of reinforcement learning and deep learning, deep reinforcement learning has unlocked the potential to navigate complex scenarios and derive optimal strategies, thereby revolutionizing problem-solving across diverse domains. DRL is also more suitable as a training method for the tasks of this paper for the following reasons:

- DRL obviates the necessity for labeled raw data, making it especially useful in cases where there are fewer microfluidic chip designs available compared to elec-

tronic counterparts. Insufficient training datasets often result in underwhelming performance in machine learning models or neural networks.

- The lack of a clear-cut model for the "ideal" microfluidic chip design poses a significant challenge. While experts in chemistry or biology can expertly guide the choice of microfluidic components and their logical arrangement, the tasks of component placement and routing often rely on heuristic methods, which can lead to less-than-optimal results. This situation presents an opportunity for improvement.
- The goals of automating microfluidic chip design vary based on specific situations. One scenario might prioritize compact layout optimization, while another could emphasize well-aligned routing with restrictions. By adjusting reward ratios smartly, these shifts in design objectives can be effectively managed, making the solution adaptable.

3.3.1 Deep Deterministic Policy Gradients (DDPG) Algorithm

The Deep Deterministic Policy Gradients (DDPG) algorithm is a popular and effective reinforcement learning algorithm that combines ideas from both Q-learning and policy gradient methods. DDPG is specifically designed to handle continuous action spaces in continuous state spaces, making it well-suited for problems where actions are not discrete but rather have a wide range of possible values [33, 22].

DDPG extends the original actor-critic architecture by utilizing two separate neural networks as shown in Figure 3.3: an Actor-network that directly maps states to actions, and a Critic network that estimates the Q-values of state-action pairs. This separation of concerns allows DDPG to learn deterministic policies, making it particularly effective in continuous control tasks.

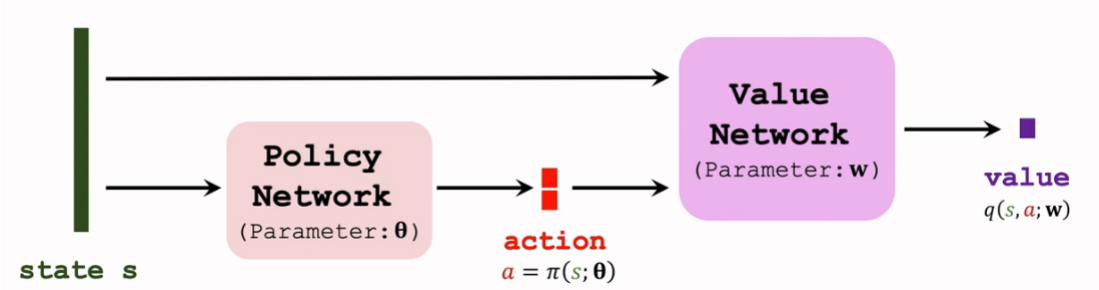

Figure 3.3: The network architecture of the DDPG algorithm [34].

One of the key features of DDPG is its use of target networks to stabilize the learning

process. The target networks are periodically updated to provide more stable Q-value estimates and reduce the issues of overestimation or divergence that can occur during training. This process involves maintaining a separate set of target networks that are updated using a soft update mechanism, which helps to improve the stability and convergence of the algorithm.

In addition, DDPG employs an experience replay buffer, similar to DQN, to store and sample experiences in a way that reduces correlations between consecutive samples. This improves the stability and efficiency of learning by breaking the temporal correlations present in consecutive state transitions.

3.3.2 Value Network

In the DDPG algorithm, the Value Network is a crucial component used for estimating the expected cumulative rewards, also known as the Q-values. The Value Network helps the agent assess the quality of a specific state-action pair by predicting the expected total reward that can be achieved from that pair.

The Value Network takes both the current state and the chosen action as inputs and outputs a single scalar value, representing the Q-value. The Q-value indicates the expected cumulative reward that the agent can achieve by following a specific action from a given state.

The architecture of the Value Network typically consists of a few fully connected layers, which process the combined information of the state and action to produce the Q-value. The network is trained using the Bellman equation, which helps it to learn accurate Q-value estimates over time.

Here is the implementation of the Value Network in PyTorch:

```
import torch
import torch.nn as nn
import torch.nn.functional as F

class ValueNetwork(nn.Module):
    def __init__(self, num_inputs, num_actions, hidden_size, init_w=3e-3):
        super(ValueNetwork, self).__init__()

        self.linear1 = nn.Linear(num_inputs + num_actions, hidden_size)
        self.linear2 = nn.Linear(hidden_size, hidden_size)
        self.linear3 = nn.Linear(hidden_size, 1)

        self.linear3.weight.data.uniform_(-init_w, init_w)
        self.linear3.bias.data.uniform_(-init_w, init_w)
```

```
def forward(self, state, action):
    if len(state.shape) < 2:
        state = state.unsqueeze(0)
    state = state.to(device) # Move input data to the same device as the model
    action = action.to(device) # Move action to the same device
    x = torch.cat([state, action], -1)
    x = F.relu(self.linear1(x))
    x = F.relu(self.linear2(x))
    x = self.linear3(x)
    return x
```

In this implementation, ***num_inputs*** represents the dimensionality of the state space, num_actions represents the dimensionality of the action space, and ***hidden_size*** determines the number of neurons in the hidden layers of the network. The forward method takes a state tensor and an action tensor as inputs, concatenates them, and processes them through the network to produce the Q-value.

The weights and biases of the final layer ***(linear3)*** are initialized using uniform random values scaled by ***init_w***.

3.3.3 Policy Network

Unlike the conventional DDPG algorithm that only deals with continuous action space, here we propose a modified DDPG algorithm with a hierarchical policy network. The construct of such a network is illustrated as in Figure 3.4

The policy network is designed to handle both high-level and low-level policies hierarchically. It consists of three components: ***LowLevelPolicy***, ***HighLevelPolicy***, and ***PolicyNetwork***.

- ***LowLevelPolicy***: This component takes the current state as input and generates low-level action parameters. It uses fully connected layers to process the input state and produces continuous actions using the tanh activation function.
- ***HighLevelPolicy***: This component takes the current state as input and generates high-level action probabilities. It employs linear layers followed by layer normalization and ReLU activation to produce a probability distribution over the available high-level actions.
- ***PolicyNetwork***: This is the main policy network that integrates the high-level and low-level policies. It takes the state as input and selects a high-level action based on the high-level policy. Then, it uses the selected high-level action to determine

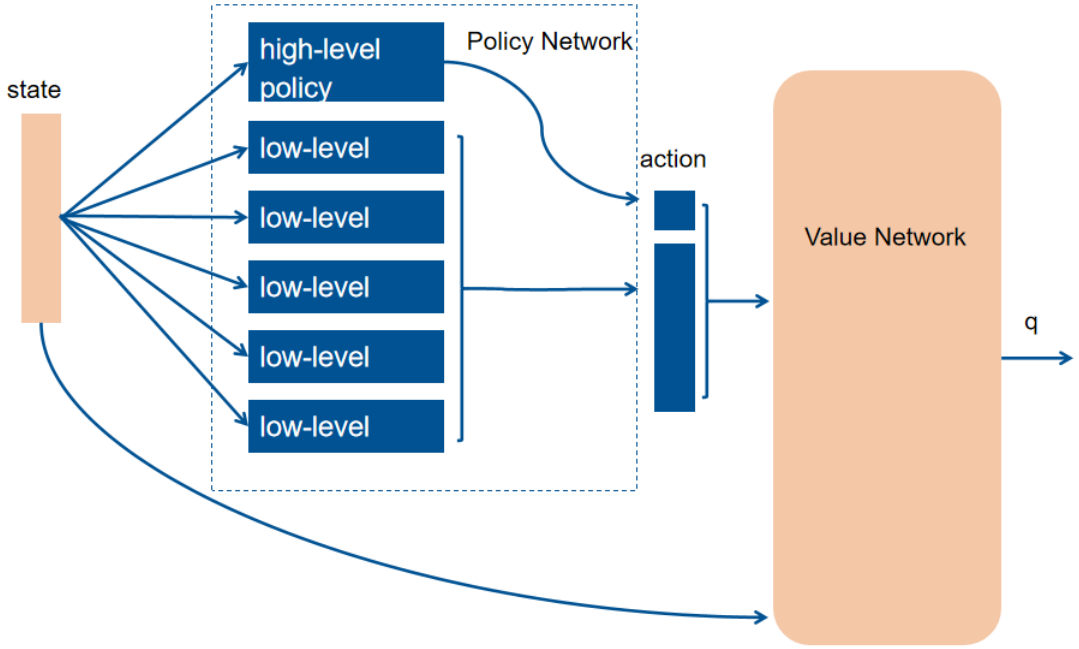

Figure 3.4: The network architecture of the DDPG algorithm with hierarchical policy network

which specific low-level policy network to use for generating the final action. It concatenates the high-level action with the low-level action parameters to form the complete action.

```
class PolicyNetwork(nn.Module):
    def __init__(self, state_dim, num_low_level_policies, hidden_size...):
        super(PolicyNetwork, self).__init__()
        self.high_level_policy = HighLevelPolicy(state_dim, num_low_level_policies)
        self.low_level_policies = nn.ModuleList()
        self.low_level_policies.append(LowLevelPolicy(state_dim, 4, hidden_size))
# for action comp placement
        self.low_level_policies.append(LowLevelPolicy(state_dim, 4, hidden_size))
# for action add node
        self.low_level_policies.append(LowLevelPolicy(state_dim, 2, hidden_size))
# for action delete node
        self.low_level_policies.append(LowLevelPolicy(state_dim, 1, hidden_size))
# for action flip comp layer
        self.low_level_policies.append(LowLevelPolicy(state_dim, 1, hidden_size))
# for action flip conn layer
        self.num_low_level_policies = num_low_level_policies
        self.epsilon = epsilon
        def forward(self, state):
            ...
        def get_action(self, state):
            ...
```

Inside the constructor:

- ***high_level_policy***: This creates an instance of the HighLevelPolicy class, which represents the high-level policy for decision-making based on the input state.
- ***low_level_policies***: This creates a list of instances of the LowLevelPolicy class, each representing a different low-level policy for specific actions (e.g., component placement, adding node, deleting node, etc.).
- ***num_low_level_policies***: This stores the total number of available low-level policies.
- ***epsilon***: This stores the exploration factor for action selection during training.

The ***forward*** method within the ***PolicyNetwork*** class is used for performing forward propagation in a deep reinforcement learning model. It takes an input state and

generates corresponding actions based on the trained neural network policies. Here's a breakdown of its main steps and functionalities:

- Preprocess Input State: This part converts the input state into a PyTorch tensor (if it's not already) and moves the state to the specified device (e.g., GPU).
- Determine Batch Size: This part determines the batch size as the first dimension of the state tensor
- Generate High-Level Policy: This part processes the input state using the high-level policy model (***high_level_policy***) to generate a high-level action. It also applies the softmax operation to the high-level action to obtain probabilities of different high-level choices (***high_level_choice_prob***).
- Generate Low-Level Policies: For each low-level policy, the corresponding low-level policy model is used to generate parameters for the low-level actions. The generated low-level action parameters are then concatenated into a tensor ***low_level_action***.
- Combine High-Level and Low-Level Actions: This part combines the high-level action and the tensor of low-level action parameters to create an overall action vector action.
- Format and Expand: If the action vector is not two-dimensional (i.e., contains additional dimensions), this part of the code will expand each one-dimensional tensor to two dimensions and concatenate them along the first dimension to create a two-dimensional batch tensor ***action_batch_tensor***. This step is necessary since the forward method should be capable of dealing with both a single state or a batch of input states when processing experience replay or calculating the loss method.
- Return the Result: The final step of the method returns the generated action batch tensor ***action_batch_tensor***, which contains information about both high-level and low-level actions.

The get_action method is responsible for generating an action based on the policy network's output. It also incorporates exploration by randomly selecting high-level actions with a certain probability (***epsilon***).

3.3.4 Experience replay

The ***ReplayBuffer*** class is a key component in the DDPG algorithm that implements an experience replay mechanism. Experience replay involves storing and randomly

sampling past experiences to break the temporal correlations in the training data, which improves the stability and efficiency of the learning process.

- ***__init__(self, capacity)***: This method initializes the replay buffer with a specified capacity to store experiences.
- ***push(self, state, action, reward, next_state, done)***: This method adds a new experience tuple to the buffer, containing the current state, action taken, reward received, next state, and a flag indicating if the episode is done.
- ***sample(self, batch_size)***: This method randomly samples a batch of experiences from the buffer. Returns a batch of states, actions, rewards, next states, and done flags.
- ***__len__(self)***: This method simply returns the current size of the replay buffer.

The use of experience replay can have many advantages:

- Breaking Temporal Correlations: In a typical online learning setting, where an agent learns from consecutive experiences, the data can be highly correlated temporally. This correlation can lead to instability during training and hinder the convergence of the learning algorithm. Experience replay breaks these temporal correlations by storing a diverse set of past experiences and randomly sampling from them. This helps in reducing the noise and variance in the learning process, leading to more stable updates.
- Data Efficiency: Experience replay allows the agent to reuse past experiences multiple times for learning. In complex environments, interactions with the environment can be time-consuming and resource-intensive. By reusing experiences, the agent can learn more from each interaction, making better use of the collected data. This is especially valuable when the amount of real interaction with the environment is limited or expensive.
- Sample Efficiency: Reinforcement learning algorithms, including DDPG, often require a large number of samples to effectively learn good policies. Experience replay helps make better use of collected samples by reusing them for multiple updates. This can significantly improve the sample efficiency of the learning algorithm, enabling the agent to learn a good policy with fewer interactions with the environment.
- Mitigating Catastrophic Forgetting: Catastrophic forgetting occurs when the agent forgets previously learned behaviors as it updates its policy based on new

experiences. Experience replay can help mitigate this issue by interleaving updates with diverse past experiences. This ensures that the agent retains knowledge about different states and actions, preventing it from forgetting valuable lessons learned earlier.

- Improving Exploration: During learning, an agent needs to explore the environment to discover the optimal policy. Experience replay can enhance exploration by allowing the agent to revisit past states and learn from them. This is especially helpful when the agent is encountering rare or challenging situations that are critical for learning but might occur infrequently during online interaction.
- Gradient Stability: When training deep neural networks, gradient stability is crucial for efficient learning. Experience replay provides a more stable gradient signal for updating the network weights by avoiding rapid changes in the training data distribution. This stability contributes to smoother and more consistent updates during training.

Overall, experience replay plays a pivotal role in enhancing the stability, efficiency, and effectiveness of reinforcement learning algorithms like DDPG. By breaking temporal correlations, improving data and sample efficiency, mitigating catastrophic forgetting, aiding exploration, and ensuring gradient stability, experience replay helps the agent learn more effectively and converge to better policies in complex and dynamic environments.

3.3.5 DDPG Nework

the DDPG class initializes the various components and hyperparameters of the DDPG algorithm. It sets up the neural networks for the value method (***value_net***), and the policy method (***policy_net***), as well as their corresponding target networks (***target_value_net*** and ***target_policy_net***).

The target networks are initially set to match the parameters of the online networks. The class also sets up the optimizers for both the value and policy networks, along with their respective learning rates. The ***value_criterion*** is defined as the Mean Squared Error (MSE) loss, which is commonly used in value method optimization.

Finally, an instance of the ***ReplayBuffer*** class is created as the experience replay buffer to store and sample past experiences for training.

3.3.6 Target Networks

In the DDPG (Deep Deterministic Policy Gradient) algorithm, the use of target networks is a crucial component to stabilize and improve the training process. Target networks

serve two main purposes: to provide stable target values for the value method during training and to mitigate the issues of overestimation of Q-values in the Q-learning update. The use of target networks has the following benefits:

- Stable Target Values for Value Method: During the training of the ***value*** method (critic network), the target values are calculated using the Bellman equation, which involves estimating the expected future reward based on the next state and the action chosen by the policy. However, using the same network to estimate both current and future values can lead to a moving target problem, making the training process unstable and slow. To address this issue, target networks are introduced. The target value network (***target_value_net***) is used to calculate the target Q-values during training, providing a more stable and consistent estimation of future rewards. By decoupling the target network from the online network, the learning process becomes more reliable and the convergence is improved.
- Mitigating Overestimation of Q-values: Traditional Q-learning algorithms, such as DQN, can suffer from overestimation of Q-values due to the use of the max operator in the Bellman equation. This overestimation can lead to unstable and suboptimal training. In DDPG, by introducing target networks, the overestimation issue is alleviated. The target network's parameters are updated slowly with a soft update mechanism, which means the target network follows a weighted average of its parameters and the online network's parameters. This soft update process helps to smooth out the learning process and reduces the likelihood of overestimating Q-values.

In summary, target networks in DDPG play a vital role in stabilizing the training process by providing stable target values for the value method and mitigating the overestimation of Q-values. By using target networks, the DDPG algorithm becomes more robust, reliable, and capable of effectively learning optimal policies in continuous action spaces.

3.3.7 Update Method DDPG

The ***ddpg_update*** method is responsible for updating the policy and value networks in the DDPG algorithm using the sampled experiences from the replay buffer. This method involves several steps to perform the necessary updates for both the policy (actor) and value (critic) networks. Here are some main steps in the update method:

- Sample from the Replay Buffer: Random samples of states, actions, rewards, next states, and done flags are drawn from the replay buffer. These samples are

used to update the networks based on the experiences collected during previous interactions with the environment.

- Compute Policy Loss: The policy loss is calculated based on the value network's evaluation of the current policy's actions. For each batch element, the value network is used to evaluate the value of the current state-action pair. The policy loss is the negative mean of these value evaluations, aiming to maximize the expected cumulative reward.
- Compute Expected Value and Value Loss: The expected value of the current state is computed using the target policy network to select the next action and the target value network to estimate the future value. The Bellman equation is used to calculate the expected value, taking into account the reward and whether the episode is done. The value loss is then computed using the mean squared error between the value predicted by the value network and the expected value.
- Update Policy Network: The policy network (actor) is updated by performing backpropagation through the policy loss and updating the policy network's parameters using the policy optimizer.
- Update Value Network: The value network (critic) is updated by performing backpropagation through the value loss and updating the value network's parameters using the value optimizer.
- Soft Target Network Updates: To stabilize training and reduce target value oscillations, a soft update is performed on the target value and policy networks. The parameters of the target networks are updated as a weighted average of their current values and the values of the online networks. This helps in providing more stable and slowly changing target values.

The ***ddpg_update*** method serves as the core of the DDPG algorithm, responsible for continuously updating the policy and value networks based on experiences stored in the replay buffer. These updates are essential for learning an effective policy in continuous action spaces and addressing the instability and overestimation challenges often encountered in reinforcement learning algorithms.

3.3.8 Update Method PADDPG

The main difference between the update methods of the original DDPG (Deep Deterministic Policy Gradient) algorithm and the PADDPG (Parameterized Action Deep Deterministic Policy Gradient) algorithm lies in how they update the policy (actor)

network. Here's a comparison of the two update methods you provided to highlight this difference:

Original DDPG Update Method:

```
def ddpg_update(self):
    # ... (sample experiences and data preparation)
    # Compute TD target and value estimates
    next_action = self.target_policy_net(next_state)
    target_value = self.target_value_net(next_state, next_action.detach())
    expected_value = reward + (1.0 - done) * self.gamma * target_value
    expected_value = torch.clamp(expected_value, self.min_value, self.max_value)
    # Compute value and policy losses
    value = self.value_net(state, action)
    value_loss = (weights * self.value_criterion(value, expected_value.detach())).mean()
    # Update policy and value networks
    # ... (calculate actor loss and update policy network)
    # ... (update value network)
    # ... (soft target network updates)
    # ... (update priorities in replay buffer)
    return value_loss
```

```
def ddpg_update(self):
    # ... (sample experiences and data preparation)
    # Compute TD target and value estimates
    next_action = self.target_policy_net(next_state)
    target_value = self.target_value_net(next_state, next_action.detach())
    expected_value = reward + (1.0 - done) * self.gamma * target_value
    expected_value = torch.clamp(expected_value, self.min_value, self.max_value)
    # Compute value and policy losses
    value = self.value_net(state, action)
    value_loss = (weights * self.value_criterion(value, expected_value.detach())).mean()
    # Calculate gradients from critic and invert them
    # ... (calculate Q_val and gradients)
    # ... (invert gradients and combine with actor gradients)
    # ... (calculate out_loss and apply backward on policy network)
    # Update policy network using out_loss
    self.policy_optimizer.zero_grad()
    out_loss.backward()
    self.policy_optimizer.step()
    # Update value network
```

```
# ... (update value network)
# ... (soft target network updates)
# ... (update priorities in replay buffer)
return value_loss, out_loss
```

The main difference is in how the policy (actor) network is updated in the PADDPG update method compared to the original DDPG update method:

1. DDPG:
    - In the original DDPG, the policy (actor) network is updated using a gradient calculated from the value network's loss with respect to the policy's actions.
    - The policy network update involves calculating an actor loss and applying gradient descent on the policy network using the actor loss.
2. PADDPG:
    - In the PADDPG, the policy (actor) network update is more intricate and involves the concept of inverting gradients and combining them with the actor gradients.
    - The gradients from the critic (value network) are inverted and combined with gradients from the actor network. These combined gradients are then used to update the policy network.
    - The process includes calculating Q-values, inverting gradients, and applying gradient descent to the policy network.

The PADDPG update method brings a fresh perspective to policy network updates by integrating inverted gradients, setting it apart from the conventional DDPG update mechanism. This distinct update process is tailored to improve the learning and convergence of the policy network within the framework of microfluidic chip design optimization.

3.4 Training

The training process of the PADDPG algorithm involves iteratively interacting with the environment, collecting experiences, and updating the policy and value networks using the DDPG updates. This chapter describes the main training loop, hyperparameters, and visualization of the training progress.

3.4.1 Training Setup

The training process of the PADDPG algorithm is structured as follows:

- Environment Initialization: The custom ***LayoutEnv*** is instantiated with specific parameters, such as the maximum number of components, ports per component, connections, nodes per connection, and free ports. The environment is also configured for visualization.
- Network Initialization: The policy and value networks are initialized with appropriate input and output dimensions, as well as hidden dimensions. The target policy and value networks are also created as separate instances, and their initial weights are synchronized with the online networks.
- Replay Buffer Setup: An instance of the ***ReplayBuffer*** class is created to store and manage experiences for experience replay.

3.4.2 Training Loop

The main training loop consists of interacting with the environment, collecting experiences, and updating the networks. The loop runs for a specified number of episodes.

- Episode Initialization: For each episode, the environment is reset, and the initial state is obtained.
- Exploration Strategy: An exploration strategy is employed using an epsilon-greedy approach. Initially, a higher epsilon value is used to encourage exploration, and then it gradually decreases over episodes.
- Step through Environment: Within each episode, the agent interacts with the environment for a maximum number of steps. The agent selects actions using the policy network's ***get_action*** method.
- Experience Collection and Replay: The agent collects experiences in the form of (***state, action, reward, next_state, done***) tuples and stores them in the replay buffer. If the buffer contains enough samples, a DDPG update is performed.
- DDPG Update: The DDPG update is executed by sampling a batch of experiences from the replay buffer. The policy and value networks are updated based on these experiences, aiming to improve the agent's performance.
- Tracking Rewards: The episode reward is calculated as a weighted sum of the current reward and the cumulative episode reward. This helps the agent consider both immediate and long-term rewards.

- Stopping Conditions: The episode terminates if the reward reduction criterion is met. If the reward decreases too frequently, it indicates the agent's inability to make progress, and the episode is halted.
- Visualization and Saving: Periodically, the rewards are plotted to visualize the training progress. The trained policy and value networks are also saved for future use.

3.4.3 Hyperparameters

Several hyperparameters influence the training process of the PADDPG algorithm:

- ***max_episode***: The maximum number of episodes for training.
- ***max_steps***: The maximum number of steps per episode.
- ***batch_size***: The size of the batch used for DDPG updates.
- ***gamma***: The discount factor for future rewards.
- ***soft_tau***: The factor for soft target network updates.
- ***epsilon***: The exploration rate for the epsilon-greedy strategy.
- ***replay_buffer_size***: The capacity of the replay buffer.
- ***value_lr***: The learning rate for the value network optimizer.
- ***policy_lr***: The learning rate for the policy network optimizer.

3.4.4 Training Visualization

The training progress is visualized by plotting the episode rewards, loss of value network, and loss of policy network. A scatter plot is used to show individual episode rewards, and a red dashed line represents the smoothed reward trend. The visualization provides insights into the learning progress and convergence of the algorithm. The illustration of these results is in the next chapter.

3.5 Model Convergence Strategies

During the training process of the proposed PADDPG algorithm for microfluidic chip design, several strategies were employed to facilitate rapid convergence and enhance the learning efficiency of the neural networks. These strategies were meticulously

designed to optimize the exploration-exploitation trade-off and to mitigate potential issues related to high-dimensional state spaces and complex reward landscapes. This section outlines the specific approaches adopted to expedite the convergence of the model.

3.5.1 Weighted Experience Replay

To enhance the learning process, a weighted experience replay mechanism was integrated into the training pipeline [35]. By assigning different weights to experiences in the replay buffer, the algorithm prioritizes samples that are likely to provide valuable learning signals. This approach effectively guided the neural networks towards focusing on important experiences, thereby accelerating the convergence rate.

3.5.2 Principal Component Analysis (PCA) for Dimensionality Reduction

The inherent complexity of microfluidic chip design spaces often presents challenges in learning from high-dimensional state representations. In reinforcement learning a common solution is feature engineering [36, 37]. Additionally, Principal Component Analysis (PCA) was applied as a feature engineering technique to reduce the dimensionality of the input state space [38]. By projecting the state vectors onto a lower-dimensional subspace capturing the most informative components, the neural networks were able to process more informative inputs, leading to improved learning efficiency.

3.5.3 Adaptive Learning Rate Scheduling

The learning rate is a crucial hyperparameter that significantly influences the optimization process. In this endeavor, a dynamic learning rate schedule was introduced, wherein the learning rate is gradually reduced over successive iterations [39]. This tailored learning rate adaptation facilitated a smoother optimization trajectory, preventing overshooting and ensuring stable convergence even in challenging optimization landscapes.

3.5.4 Iterative Exploration-Decay Strategy

Balancing exploration and exploitation is a fundamental challenge in reinforcement learning. To address this, an iterative exploration-decay strategy was often implemented, wherein the exploration rate (epsilon) decreases progressively with the number of iterations [40, 41]. This strategy allowed the algorithm to transition from a more

exploratory behavior in the early stages of training to a more focused, exploitation-driven approach as the training progressed, ultimately aiding in swift convergence.

3.5.5 Network Architecture Adaptation

The architecture of neural networks plays a pivotal role in determining their representational power. In pursuit of convergence optimization, variations in the depth and complexity of the neural network architectures were explored [42, 43]. By experimenting with different network depths and layer configurations, the algorithm was fine-tuned to better capture the underlying dynamics of the microfluidic chip design process.

3.5.6 Neural Network Parameter Normalization

To stabilize and accelerate the convergence process, the parameters of the neural networks were subjected to normalization [44]. This normalization procedure standardized the parameter ranges, preventing issues associated with vanishing or exploding gradients. Consequently, the training dynamics were improved, leading to a more efficient convergence process.

Incorporating these model convergence strategies collectively has contributed to a substantial enhancement in the efficiency and stability of the proposed PADDPG algorithm during the training phase. By thoughtfully tailoring these strategies to the unique challenges posed by microfluidic chip design optimization, the algorithm demonstrated improved convergence rates, facilitated learning in high-dimensional spaces, and effectively harnessed the power of deep reinforcement learning to achieve optimal chip designs.

4 Result and Discussion

In this chapter, the outcomes of applying the PADDPG algorithm to the microfluidic chip design optimization problem will be presented and analyzed. The discussion will delve into the implications of the results, drawing connections to the key strategies employed in the model and their impact on convergence, performance, and efficiency.

4.1 Performance Metrics and Evaluation

The introduction of performance metrics is a common way of evaluating the performance of trained models or designed microfluidic devices [45, 46]. This section will begin by introducing the performance metrics employed to evaluate the effectiveness of the PADDPG algorithm in microfluidic chip design optimization. These metrics may encompass quantitative measures such as chip size, connection length, component overlap, and other relevant design parameters. Furthermore, a comprehensive comparison with existing manual and heuristic-driven approaches will be conducted to showcase the advantages of the PADDPG-generated designs in terms of efficiency and optimality.

- Chip Size: One fundamental metric to evaluate is the size of the microfluidic chip. The compactness of the chip layout, as determined by the PADDPG algorithm, will be compared to manual designs. The metric will provide insights into how well the algorithm optimizes the placement of components, leading to efficient space utilization and potential cost savings in fabrication.
- Connection Length: The length of fluidic connections between various components on the chip is another pivotal indicator of design quality. Total connection length is also a very common criterion applied in many routing optimization-related problems [9, 13, 29, 47]. A comparison with heuristic and manual designs will highlight the algorithm's prowess in optimizing fluidic paths.
- Component Overlap: Overlap among microfluidic components can result in mutual interference between the components and fluidic pathways, leading to compromised chip functionality and rendering the chip non-operational as intended, due to potential cross-contamination and disruption of fluid flow channels. The algorithm's ability to mitigate component overlap will be assessed

using this metric, showcasing its potential to generate layouts that ensure isolation and improve experimental accuracy.

- Computational Efficiency: The execution time required for the PADDPG algorithm to converge to optimal will be compared with manual design processes. The computational efficiency of the PADDPG model will be illustrated as the computational time between the network's input and output using a GPU with 12GB VRAM.

4.1.1 Experimental Procedure

The experimental process aims to rigorously assess the performance of the PADDPG algorithm in the context of microfluidic chip design optimization. To achieve this, a structured experimentation framework will be established, involving the utilization of an abstract microfluidic chip environment introduced in Chapter 3. This environment will facilitate the generation of diverse initial input parameters, each associated with distinct topological structures.

The experimental procedure unfolds as follows:

Generation of Initial Parameters

In this stage, the abstract microfluidic chip environment will randomly generate initial input parameters characterized by varying topological structures. These parameters will serve as the starting point for both the neural network-based PADDPG algorithm and manual design conducted by human experts.

Application of PADDPG and Manual Design

The generated initial parameters will undergo a series of five predefined operations, as dictated by the microfluidic chip environment. Both the PADDPG algorithm and human experts will independently apply these operations to the input parameters. During this process, various performance metrics, as described in Section 6.1, will be meticulously recorded and documented for subsequent comparative analysis.

Validation and Termination Criteria

During the validation phase, an iterative approach will be employed to update the rewards associated with new actions. Specifically, a threshold value for reward updates will be established, and if the cumulative rewards obtained from the most recent five actions fall below this threshold, the ongoing episode will be concluded. This validation

mechanism ensures that the algorithm's actions align with the desired optimization objectives.

Network Operation and Policy Network Updates

When the neural network-based PADDPG algorithm is deployed to manipulate the microfluidic chip environment, solely the policy network will be invoked to produce action parameters. Importantly, during this operational phase, the parameters of the neural network will remain unaltered, effectively preserving the learned policy.

Table 4.1: Table of experiment results in performance metircs.

pins on comp. #(p_1, p_2,... p_n)	model	S(mm^2)	L(mm)	T(s)
(5, 2, 3)	Manual	282.3	168.3	>60
	DADDPG	257.1	176.2	0.7050
(3, 3, 4)	Manual	356.1	275.1	>60
	DADDPG	321.0	260.0	0.6885
(2, 2, 3)	Manual	437.9	332.2	>60
	DADDPG	403.3	301.3	0.5959
(5, 4, 3, 2)	Manual	512.2	575.4	>240
	DADDPG	480.1	585.6	1.8013
(2, 3, 3, 3)	Manual	339.6	461.1	>180
	DADDPG	302.1	497.2	1.7052
(4, 4, 5, 5)	Manual	617.5	832.3	>240
	DADDPG	540.3	776.8	1.788
(2, 2, 3, 5, 4)	Manual	423.6	621.5	>300
	DADDPG	391.7	623.7	2.021
(5, 5, 3, 4, 4)	Manual	882.4	963.6	>600
	DADDPG	601.1	1162.5	2.785

As in Table 4.1, PADDPG impressively reduces chip area through systematic exploration and component placement, enhancing space utilization. Through comparison, it is found that the layout generated by PADDPG can often make better use of the double-layer characteristics of the chip to improve the area utilization efficiency of the chip. PADDPG's connection length optimization varies across topologies, occasionally rivaling manual design. Human intuition plays a role in achieving efficient connections. Another noteworthy aspect is the computational efficiency of the PADDPG algorithm. Compared to the significant time investment required for model training (spanning

several hours or even days), the time taken by a well-trained model to generate action sequences and optimize the topology of input microfluidic chips is remarkably short, often only a matter of seconds. This efficiency enhancement is pronounced when contrasted with manual design processes.

4.2 Convergence Analysis

In this section, we delve into the convergence analysis of the PADDPG algorithm. To comprehensively assess the impact of different convergence strategies on the training process, a series of experimental steps were carefully designed. These steps involved the systematic inclusion or removal of the previously mentioned convergence strategies. By conducting these experiments, we aimed to illuminate how each strategy contributes to the convergence behavior of the PADDPG network during the training phase.

4.2.1 Convergence Metrics

To quantify the convergence performance of PADDPG under each experimental setup, we employed several key metrics, including:

- Training Curve Analysis: Monitoring the progression of the training curve over episodes to observe convergence trends.
- Reward Convergence: Analyzing the stability and consistency of the cumulative rewards obtained by the network during training.

4.2.2 Experimental Design and Evaluation

The convergence analysis experiments were structured as follows:

Baseline Experiment

The PADDPG algorithm was trained with all specific convergence strategies incorporated. This baseline served as a reference for evaluating the effectiveness of subsequent strategies. Different from the general baseline, where all the convergence strategies are not used, and then add each strategy exclusively, now the comparison with the baseline adopts the method of subtracting. Because if all the convergence strategies are not used, it is difficult for the neural network to converge, making the comparison results not informative.

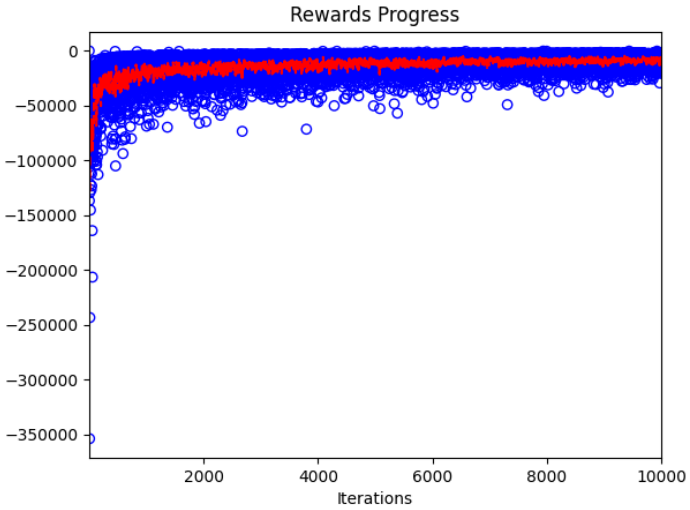

Figure 4.1: Rewards over training episode baseline.

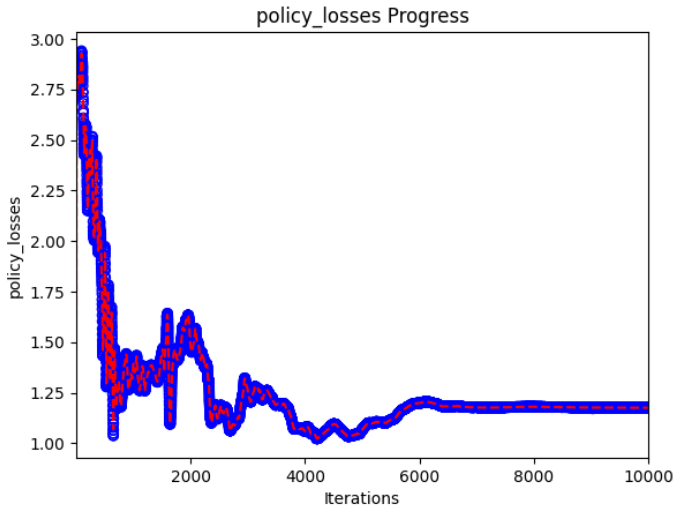

Figure 4.2: Loss of policy network over training episode baseline.

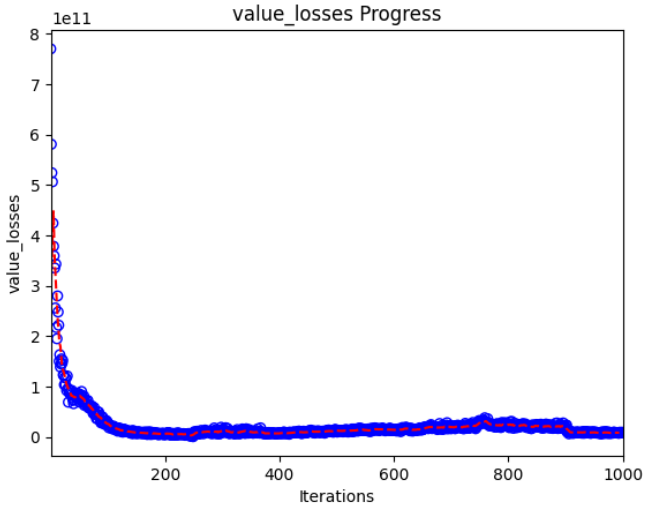

Figure 4.3: Loss of value network over training episode baseline.

From Figure 4.1, Figure 4.2 and Figure 4.3, several observations can be made. The blue hollow points represent individual training samples, while the red dashed lines represent the smoothed average values, illustrating the overall trends.

The value network converges earliest, with its loss reaching a low value after approximately 200 training iterations. Interestingly, the different convergence strategies employed in this experiment have minimal impact on the convergence behavior of the value network. This outcome can be attributed to the inherent ease of convergence of the value network itself. Consequently, for subsequent comparisons, the "loss of value net" will be excluded from consideration.

Furthermore, the behavior of the reward and the loss of the policy network are notable. During the initial 200 iterations, both the reward and the loss of the policy network exhibit rapid convergence. However, after around 6,000 iterations, the reward continues to display significant variance, and the loss of the policy network stabilizes around a non-zero constant value rather than converging to zero. This phenomenon is rooted in the fact that each training instance involves the generation of a randomly initialized topology, resulting in varying complexities. The sustained non-convergence of the loss of the policy network indicates that the agent is continually learning and adapting to the changing topologies. Interestingly, a gradual increase in the trend of the reward is noticeable, accompanied by an enhanced level of stability. This implies that the learning process is gradually improving, and the agent is making progress in achieving more favorable outcomes. The results collectively highlight the intricate

interplay between convergence, randomness in topology generation, and the agent's learning dynamics within the microfluidic chip design context.

Weighted Experience Replay

The influence of weighted experience replay was assessed by training the network with weighted experience replay and experience with equal weights. In the subsequent

Figure 4.4: Reward over training episode baseline and without weighted experience replay.

analysis, a comparison is made between the baseline and the variant where weighted experience replay is removed, focusing on the variation of reward across different episodes. In Figure 4.4, the blue hollow points represent individual training samples from the baseline, while the red dashed line signifies the average value of rewards obtained from baseline training. Additionally, the yellow hollow points correspond to individual training samples after the removal of weighted experience replay, and the green solid line represents the average reward values obtained from training without this strategy.

Observing the graph, it becomes evident that the removal of weighted experience replay results in slower convergence of the neural network. This phenomenon can be attributed to the influential role of weighted experience replay in prioritizing and emphasizing the learning from experiences that have a significant impact. When this strategy is removed, the learning process becomes less focused on crucial experiences,

leading to a gradual and delayed convergence.

Weighted experience replay serves as a mechanism to guide the learning process by assigning higher importance to experiences that provide valuable insights and knowledge. By selectively emphasizing certain experiences, the neural network can adapt more effectively to complex and critical scenarios, ultimately accelerating convergence. The observed difference in convergence rates between the baseline and the variant without weighted experience replay underscores the importance of this strategy in facilitating more efficient learning and achieving better outcomes in microfluidic chip design.

PCA-based Feature Engineering

To understand the impact of dimensionality reduction through PCA, the algorithm was trained with and without PCA-based state representations. Figure 4.5 shows the

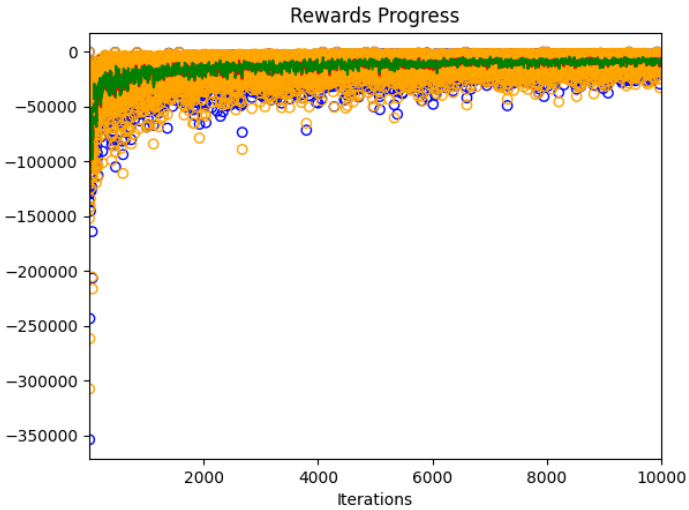

Figure 4.5: Reward over training episode baseline and without PCA.

comparison involves the baseline and the variant without the application of PCA-based feature engineering, focusing on the reward variations across episodes. Similar to the representation of the weighted experience replay comparison, the color scheme for data points remains consistent: red for the baseline, and green for the variant without the convergence strategy. In this context, the absence of PCA-based dimensionality reduction is reflected.

Upon examination of the graph, it becomes evident that the reward curve for the

variant without PCA exhibits slightly faster convergence compared to the baseline, requiring fewer iterations. However, the actual benefit brought by PCA extends beyond convergence speed. PCA introduces a reduction in the complexity of the neural network, resulting in significantly reduced iteration time during training. This dynamic represents a trade-off between network performance and computational efficiency.

By employing PCA, the network's ability to capture relevant patterns and features is preserved, even though it may converge slightly faster without PCA. Moreover, the computational advantage gained from reduced iteration time is noteworthy. When training resources are abundant or the computational power of the training environment is substantial, opting to exclude PCA-based feature engineering might be a viable choice. This trade-off decision allows practitioners to tailor their approach based on the availability of resources and their specific training objectives.

To conclude, the comparison highlights the impact of PCA-based feature engineering on convergence speed and computational efficiency. While the absence of PCA may lead to marginally faster convergence, its incorporation brings about the benefits of simplified network architecture and reduced training time. The decision to include or exclude PCA should be made considering the available resources and the desired trade-off between network performance and training efficiency.

Dynamic Learning Rate

The effect of dynamically adjusting the learning rate was explored by enabling and disabling this strategy during training. In the context of the dynamic learning rate comparison, the focus shifts to the behavior of the loss of the policy network. Similar to the previous comparisons, the analysis involves a comparison between the baseline and the variant with the Dynamic Learning Rate convergence strategy. In this case, the color scheme stays the same: red for baseline, and green for the variant without the convergence strategy.

Figure 4.6 depicts the loss of the policy network and provides crucial insights into the impact of dynamic learning rate adjustments. Observations from the graph reveal that maintaining a fixed learning rate can lead to undesirable oscillations in the loss curve. These oscillations manifest as fluctuations in the loss values over successive iterations, resulting in a lack of consistent and smooth convergence.

The adverse effects of oscillating loss are multifaceted. First and foremost, such fluctuations signify instability in the training process. The inability to converge steadily impedes the optimization process and could potentially result in prolonged training times. Moreover, oscillations could indicate that the learning rate is too high, causing the optimization process to overshoot the optimal parameter values and preventing the network from converging effectively. This phenomenon is particularly concerning

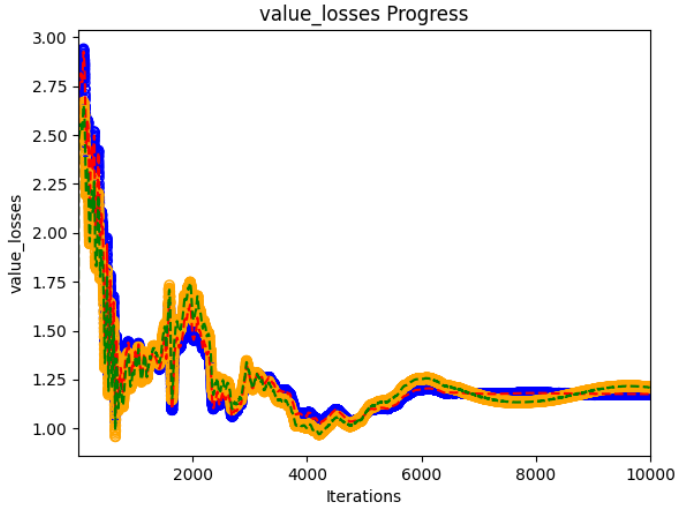

Figure 4.6: Loss of policy network over training episode baseline and without dynamic learning rate.

because it hinders the network's ability to learn and generalize patterns from the training data.

Furthermore, oscillating loss values can hinder the learning dynamics of the policy network. The erratic updates to the network's parameters can lead to a lack of coherent gradient directions, thereby slowing down the convergence process. The network might struggle to navigate toward the optimal parameter space due to the unpredictable changes in loss.

Decaying Exploration Rate

The decaying exploration rate strategy was individually incorporated to analyze its impact on convergence. In the subsequent analysis, we shift our attention to comparing the effects of a decreasing exploration rate (baseline) with a fixed exploration rate on the loss of the policy network. As previously, Figure 4.7 differentiates between the baseline (red) and the fixed exploration rate variant (green).

Upon observing the graph detailing the loss of the policy network, a noteworthy trend becomes evident: During the early iterations, the loss associated with the fixed exploration rate remains notably lower than that of the decreasing exploration rate. This observation raises intriguing questions about the underlying dynamics.

The phenomenon of lower loss with a fixed exploration rate at initial iterations

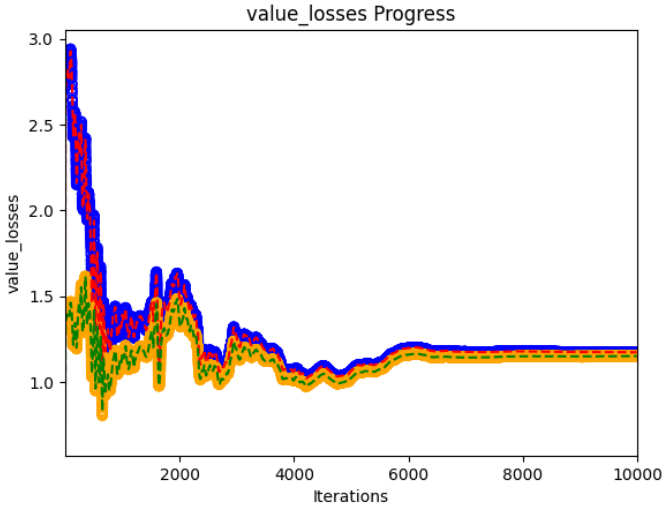

Figure 4.7: Loss of policy network over training episode baseline and without decaying exploration rate.

can be attributed to the exploration-exploitation trade-off inherent in reinforcement learning. A fixed exploration rate ensures a consistent and steady level of exploration throughout the training process. As a result, during the initial stages of training, when the network's parameters are relatively far from optimal values, a higher level of exploration aids in discovering a broader range of state-action pairs. This enhanced exploration can lead to more efficient policy updates, contributing to lower policy loss in the short term.

However, as training progresses and the policy network becomes increasingly refined, the benefits of exploration diminish. At this stage, the network's focus shifts from exploration to exploitation—i.e., refining its learned policy based on the accumulated experience. The decreasing exploration rate variant excels in this aspect. By gradually reducing the exploration rate over time, the network is guided to rely more on its learned policy, favoring exploitation. This strategy allows the network to converge more effectively toward a robust and optimal policy.

From the analysis of Figure 4.7, we can observe the effect the decreasing exploration rate strategy brings to the training process. This advantage lies in its ability to strike a balance between exploration and exploitation. While a fixed exploration rate may provide early gains in terms of lower loss due to consistent exploration, it lacks the adaptability to transition smoothly to exploitation as the policy network matures. The decreasing exploration rate strategy addresses this limitation by gradually shifting

focus towards exploitation, enabling the network to converge to a higher-quality policy throughout training. The early advantage of a fixed rate in terms of lower policy loss is offset by its reduced adaptability. The decreasing exploration rate strategy demonstrates its strength in enabling smoother transitions between exploration and exploitation, ultimately leading to more effective policy convergence.

Network Architecture Alteration

The algorithm's training performance was evaluated by modifying the architecture of the neural network. The investigation focuses on the low-level policy network, with a comparison between a baseline configuration consisting of 7 layers and an alternative configuration with a reduced depth of 3 layers. As before, the color-coded representation aids in distinguishing the baseline (red) and the altered architecture (green) for clarity.

Upon examining the reward-episode graph, a discernible contrast emerges between the two architectural configurations. The baseline network, comprising 7 layers, exhibits a clear convergence pattern, steadily improving the reward across episodes. In contrast, the network with a shallower architecture of 3 layers encounters considerable difficulty in achieving convergence.

The primary rationale behind this phenomenon lies in the complexity of representation that each architecture can capture. Deeper networks possess a higher capacity to model intricate relationships within the environment and learn more intricate decision-making policies. In the context of our PADDPG algorithm, a 7-layer low-level policy network is adept at encoding the complexities inherent in microfluidic chip design, facilitating accurate learning and effective policy convergence.

Conversely, reducing the network's depth to 3 layers leads to a constrained representational capacity. The diminished ability to capture intricate design nuances hampers the network's learning process. Consequently, the shallower network struggles to comprehend and generalize from the vast and intricate design space of microfluidic configurations. The resultant difficulty in capturing subtle design interactions and optimal decision-making policies renders the convergence process challenging.

In simple terms, the difference in how quickly the two network designs learn and improve highlights the importance of the network's depth in helping the learning process and policy improvement within the PADDPG framework. While the deeper network accommodates the complexity of the microfluidic design environment, enabling accurate policy optimization, the shallower architecture proves inadequate in capturing the intricate design space, hindering its ability to converge.

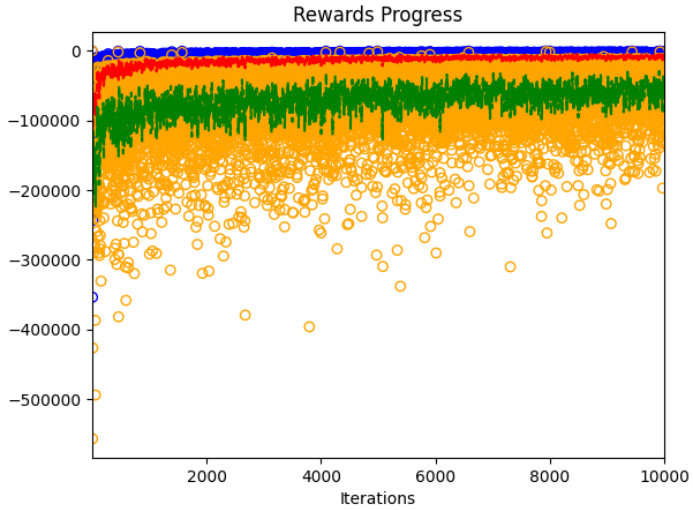

Figure 4.8: Reward over training episode baseline and change the layer number of low-level policy from 7 to 3.

Parameter Normalization

The training process was conducted with and without parameter normalization to gauge its contribution.

In the final part of our analysis, we turn our attention to the effect of parameter normalization on the convergence performance of the PADDPG algorithm. Upon incorporating parameter normalization into the neural network training process, a notable enhancement in convergence behavior becomes evident. This phenomenon can be attributed to the normalization's ability to stabilize and accelerate the learning process, thereby facilitating more efficient policy optimization.

Parameter normalization entails scaling the network's input features to a common range, ensuring that they possess similar magnitudes. This normalization step holds substantial benefits for training deep neural networks. Firstly, it mitigates the notorious vanishing and exploding gradient problems that often plague deep networks during training. By maintaining consistent input magnitudes, parameter updates are controlled, preventing overly large or tiny gradients that can impede learning and cause slow convergence.

Moreover, parameter normalization enhances the network's ability to learn across various layers. It ensures that all layers receive input within a similar range, thereby preventing any specific layer from becoming saturated or underutilized due to extreme

Figure 4.9: Reward over training episode baseline and without parameter normalization.

input magnitudes. This balanced input distribution allows each layer to contribute effectively to the learning process, facilitating the propagation of useful gradients throughout the network.

The benefits of parameter normalization are particularly pronounced in the context of the PADDPG algorithm for microfluidic chip design. The complex and high-dimensional design space of microfluidic configurations can lead to challenges in learning and policy optimization. The introduction of parameter normalization helps alleviate these challenges by promoting smoother and more efficient gradient updates.

Furthermore, parameter normalization has been empirically shown to improve the generalization ability of neural networks. This aspect is crucial for the PADDPG algorithm, as it enhances the network's capability to generalize learned policies across a wide range of microfluidic design scenarios. By promoting better generalization, parameter normalization aids the network in converging to effective policies that yield superior performance on unseen design tasks.

4.3 Design Visualization and Interpretation

This section aims to provide insights into how the algorithm's policies manifest as tangible chip layouts, emphasizing the visualization of components, connections, and layers.

The visualization of a microfluidic chip layout involves the depiction of its various elements using distinct graphical representations. Rectangles symbolize components, where solid red circles on components represent ports, and solid green circles represent freeports, indicating the points of connection between the chip and external systems. Straight or segmented lines connecting ports and freeports signify connections or channels facilitating the flow of fluids or substances. Notably, due to the hierarchical nature of the model, connections, and components of different layers are color-coded: blue for bottom-layer components, red for bottom-layer connections, green for top-layer components, and yellow for top-layer connections.

Figure 4.10, Figure 4.12 and Figure 4.14 illustrate an initial microfluidic chip layout generated randomly by the environment. This layout comprises three components, with two larger components partially overlapping due to their similar sizes. Two freeports are depicted as stacked green circles, denoting their identical initial positions. The positions of ports (red circles) are fixed relative to the components, and the connection topology remains consistent. The outermost cyan dashed line represents the chip boundary, ensuring a safe distance from all components, connections, and ports.

Following this visualization, the trained PADDPG model is introduced to optimize the initial layout. Only the policy network is utilized for action generation, while the simulation environment updates the layout based on the generated actions. This iterative process continues until a convergence criterion is met. Figure 4.11, Figure 4.13 and Figure 4.15 showcase the optimized layout achieved through this process. Notably, the optimized layout eliminates stacking penalties, minimizing both the overall component volume and connection length while preserving the proportions of individual components and the connection topology.

The actions of the policy network are logged, uncovering the sequence of steps that contributed to layout optimization. This sequence often includes actions like adjusting component layers. The resulting optimized layout can be enhanced by including specific component details, such as dimensions (e.g., mixer size), that the neural network doesn't optimize. This extra information enables the generation of JSON or STL format files, suitable for the fabrication process.

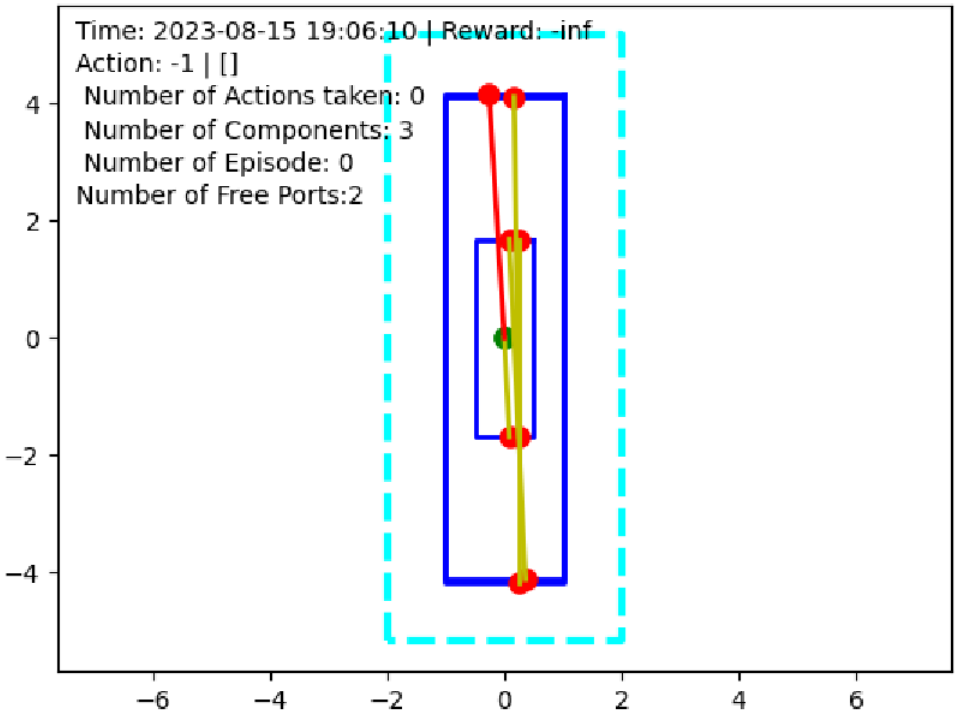

Figure 4.10: Example 1: Layout generated randomly.

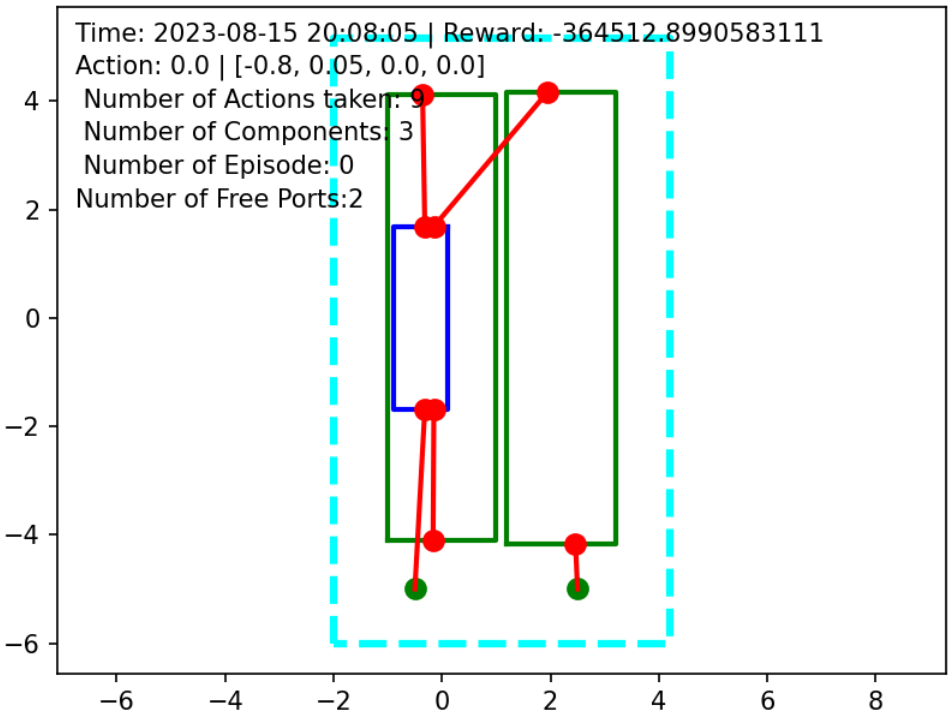

Figure 4.11: Example 1: Layout after optimization by policy network.

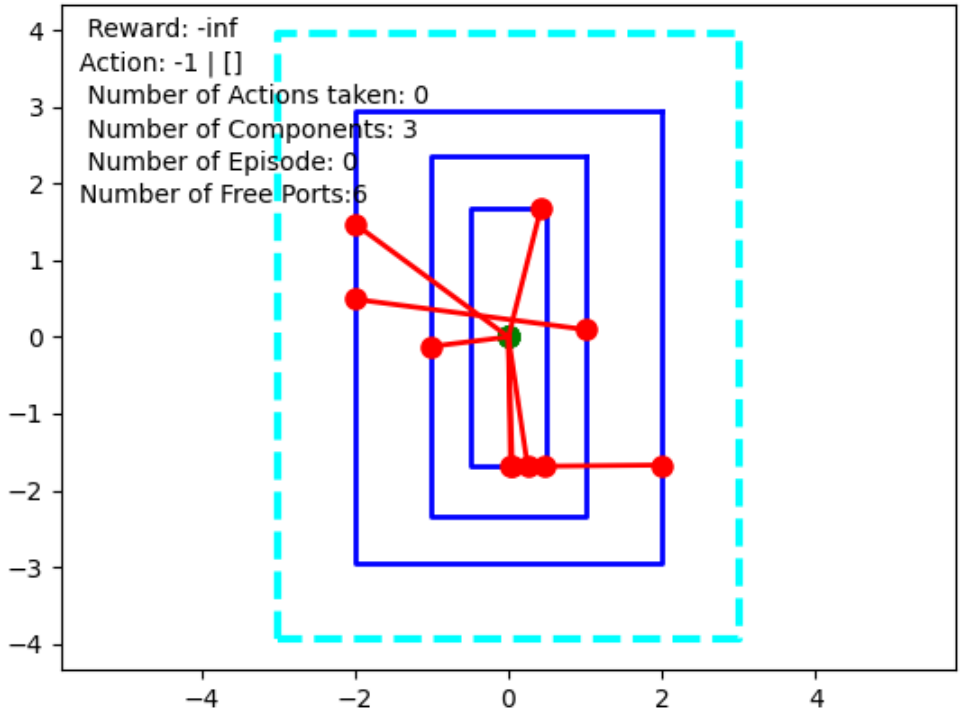

Figure 4.12: Example 3: Layout generated randomly.

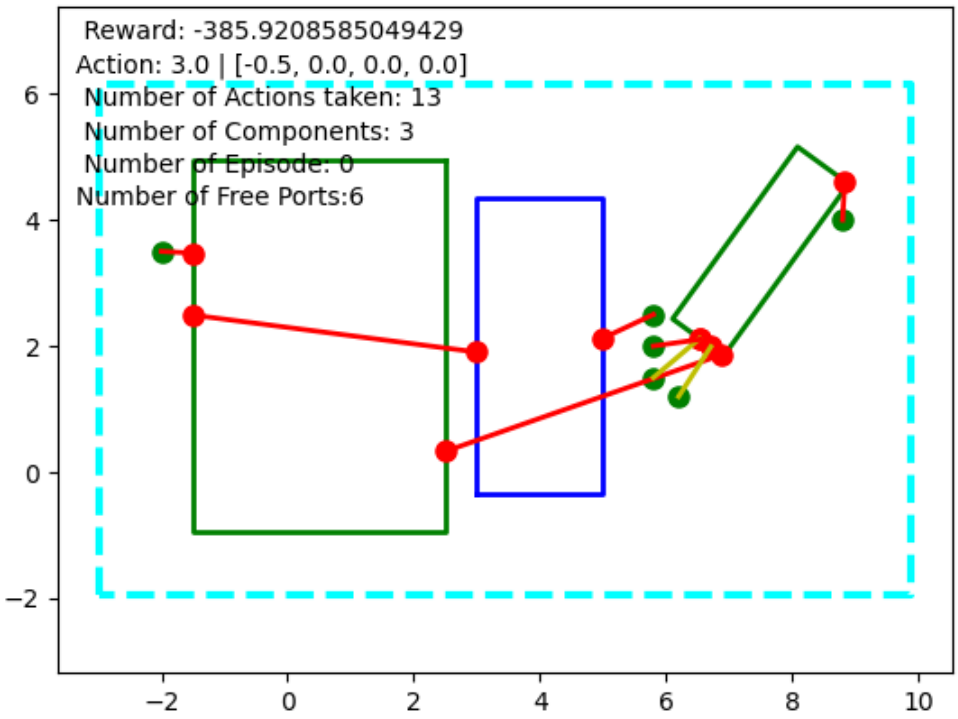

Figure 4.13: Example 3: Layout after optimization by policy network.

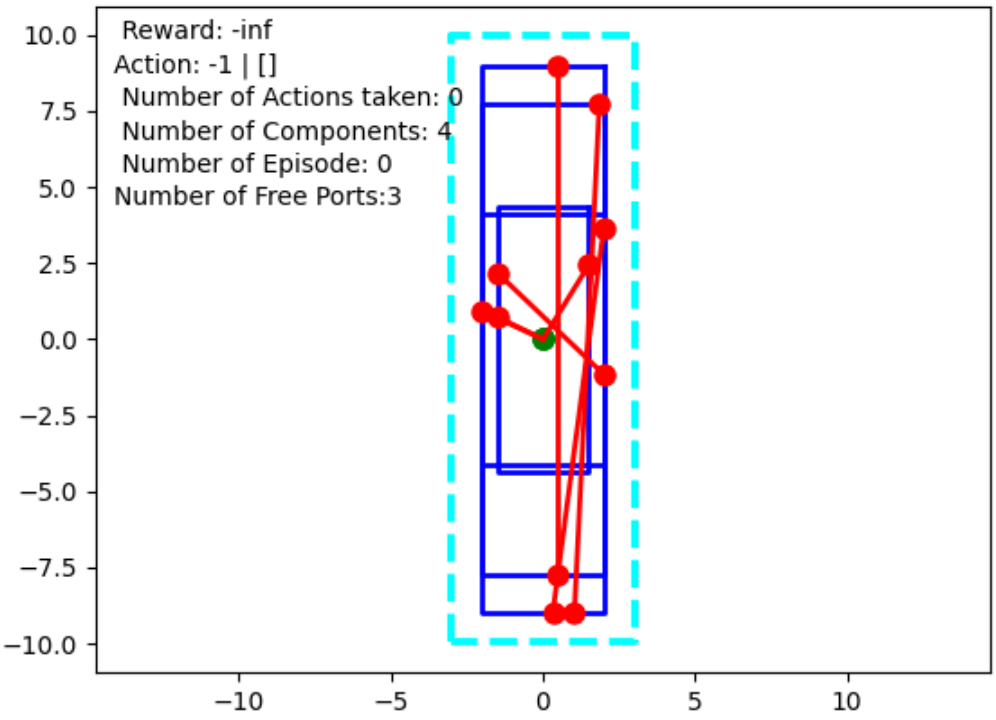

Figure 4.14: Example 3: Layout generated randomly.

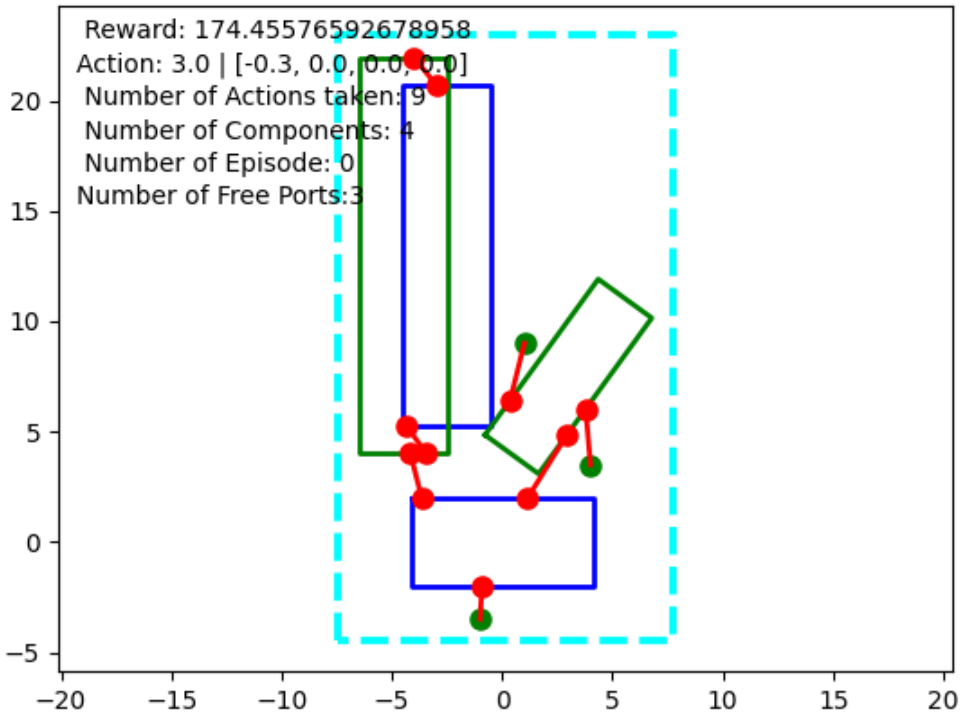

Figure 4.15: Example 3: Layout after optimization by policy network.

5 Summery and Outlook

5.1 Contributions

This research contributes by amalgamating interdisciplinary fields, overcoming data scarcity challenges through RL, simplifying the design process, introducing innovative algorithmic adaptations, and providing a versatile format for seamless integration into microfluidic chip design workflows. These contributions collectively pave the way for enhanced design automation and optimization within the realm of microfluidic chip design.

- Integration of Diverse Fields: This research bridges two promising domains, namely microfluidic chip design and deep reinforcement learning (RL), to address the scarcity of previous work that applies deep learning to the holistic layout design of microfluidic chips. By combining these disciplines, a novel and interdisciplinary approach emerges, paving the way for innovative design automation.
- Overcoming Data Scarcity with RL: Addressing the challenge of limited available data in microfluidic chip design, this study leverages RL to train an agent capable of autonomously optimizing microfluidic chip layouts. The custom reward formulation empowers the agent to learn design optimization strategies, allowing for adaptation to diverse design requirements by adjusting factors or modifying reward weights.
- Simplification of Complex Design Process: Microfluidic chip design is inherently intricate, involving a multitude of complex operations. This research simplifies the process by abstracting and simulating the chip layout as a combination of geometric shapes. By reducing the design space to five distinct operations, the complexity is significantly diminished, facilitating more effective exploration.
- Hybrid Action Space DDPG Algorithm: The utilization of a hybrid action space within the Deep Deterministic Policy Gradient (DDPG) algorithm elegantly addresses the challenge posed by the mix of discrete and continuous design actions. Through thoughtful algorithmic adaptations and exploration of various strategies, the training process is optimized to effectively navigate this complex action space.

- Universal JSON-based Input and Output: Both the input and output of the proposed model can be seamlessly transformed into a universal JSON format. This design choice facilitates integration into an end-to-end microfluidic chip design generation pipeline. The resulting layouts, enriched with component details, can be conveniently used for fabrication or further refinement.

5.2 Limitations and Future Work

While the proposed algorithm demonstrates promising results, several limitations and opportunities for future research are worth noting:

- Algorithmic Complexity: The complexity of the algorithm could be further enhanced to handle cases involving an excessive number of components or highly intricate topologies in microfluidic chip designs. Exploring techniques for optimizing computational efficiency and scalability remains an important avenue for future investigation.
- Limited Feature Consideration: Currently, the algorithm predominantly focuses on the size characteristics of microfluidic chips, neglecting other crucial attributes such as flow rate. Future iterations could integrate additional features into the reward function, enabling the optimization of more comprehensive design objectives.

There are also some promising future directions one might continue the work:

- Exploring Advanced DRL Architectures: Investigating more intricate Deep Reinforcement Learning (DRL) architectures, coupled with enhanced computational resources, holds potential for optimizing training efficiency and achieving even more sophisticated design automation.
- Combining Supervised Learning: As the dataset of microfluidic chip design examples expands, a hybrid approach involving supervised learning could be explored. This combination could harness the strengths of both supervised and reinforcement learning, potentially leading to accelerated convergence and improved design outcomes.
- Incorporating Flow Rate Optimization: Flow Rate is a criterion used in many microfluidic design optimization problems [48, 49]. Incorporating flow rate considerations into the RL reward function would offer a more comprehensive and realistic design perspective. This addition could result in microfluidic chip designs that not only minimize size but also optimize fluid dynamics performance.

- Validation with Simulation and Fabrication: The proposed designs could be rigorously validated using advanced simulation techniques such as Computational Fluid Dynamics (CFD) simulations or by physically fabricating the automatically generated chip layouts using 3D printing [50, 51, 52]. This validation process would provide deeper insights into the practical feasibility and performance of the designed microfluidic chips.

5.3 Summary

This research commenced with a thorough Literature Review, uncovering the complexities of microfluidic chip design. Existing challenges were carefully examined, leading to the exploration of innovative solutions. This paper then discusses the creation of a microfluidic chip environment abstraction. This environment was designed to encompass the intricate elements of chip components, their connections, and layers. This abstraction provided the foundation for the training of an RL agent using the PADDPG algorithm.

As the research progressed, attention shifted to convergence strategies integrated into the training process. These strategies, such as weighted experience replay, PCA-based feature engineering, dynamic learning rate, exploration rate decay, network architecture adjustments, and parameter normalization, underwent thorough analysis and testing. Each strategy's contribution to improved convergence and the algorithm's effectiveness was meticulously explored, yielding detailed insights into their combined influence. The research reached its conclusion through Experiments and Validation. Thorough experimentation and careful validation highlighted the practicality of PADDPG. Quantitative measures like chip size, connection length, and computational time were compared between PADDPG-generated designs and manually crafted ones. The convergence analysis revealed how the combined strategies affected training and results.

In summary, this paper presents a method that applies DRL in parameterized action space to the field of microfluidic chip design automation. On one hand, the paper introduces a construction approach for abstracting the microfluidic chip environment, employing simple geometric shapes to represent chip components and connections, facilitating interaction between the environment and the RL agent. On the other hand, based on DDPG, the paper modifies the originally fully connected policy network into a structured network, with a high-level policy network determining action types and a lower-level network specifying action parameters. This design effectively addresses the challenge of mixed action spaces (both discrete and continuous), and validation confirms that the trained network can interact reasonably with the abstract environment, swiftly producing optimized layouts under conditions like overlap and chip size. These

layouts can be output in JSON format for generating STL files using Flui3D, and subsequently, the microfluidic chip can be 3D printed. This work employs reinforcement learning to overcome the challenge of limited microfluidic chip samples and uses a well-designed neural network structure to tackle mixed action spaces, yielding promising results. Future work can build upon this foundation by adding or modifying reward conditions, and exploring different deep-learning algorithms and strategies, which holds the potential for achieving even better outcomes.

List of Figures

List of Tables

Bibliography

[1] G. M. Whitesides. "The origins and the future of microfluidics." In: *nature* 442.7101 (2006), pp. 368–373.

[2] K. Arulkumaran, A. Cully, and J. Togelius. "Alphastar: An evolutionary computation perspective." In: *Proceedings of the genetic and evolutionary computation conference companion*. 2019, pp. 314–315.

[3] Y. Zhang, T.-M. Tseng, and U. Schlichtmann. "Eine interaktive Design-Plattform für 3D-gedruckte mehrlagige Mikrofluidikchips mit Design-for-Manufacturing-Funktion." In: VDE/VDI-GMM MikroSystemTechnik Kongress. Oct. 2023.

[4] E. E. Tsur. "Computer-aided design of microfluidic circuits." In: *Annual review of biomedical engineering* 22 (2020), pp. 285–307.

[5] W. Thies, J. P. Urbanski, T. Thorsen, and S. Amarasinghe. "Abstraction layers for scalable microfluidic biocomputing." In: *Natural Computing* 7 (2008), pp. 255–275.

[6] H. Huang. "Fluigi: An end-to-end software workflow for microfluidic design." PhD thesis. Boston University, 2016.

[7] D. A. Gatenby and G. Foo. "Design for X (DFX): key to competitive, profitable products." In: *AT&T Technical Journal* 69.3 (1990), pp. 2–13.

[8] T.-M. Tseng, B. Li, U. Schlichtmann, and T.-Y. Ho. "Storage and caching: Synthesis of flow-based microfluidic biochips." In: *IEEE Design & Test* 32.6 (2015), pp. 69–75.

[9] K. Hu, T. A. Dinh, T.-Y. Ho, and K. Chakrabarty. "Control-layer routing and control-pin minimization for flow-based microfluidic biochips." In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 36.1 (2016), pp. 55–68.

[10] C.-X. Lin, C.-H. Liu, I.-C. Chen, D. Lee, and T.-Y. Ho. "An efficient bi-criteria flow channel routing algorithm for flow-based microfluidic biochips." In: *Proceedings of the 51st Annual Design Automation Conference*. 2014, pp. 1–6.

[11] K. Yang, H. Yao, T.-Y. Ho, K. Xin, and Y. Cai. "AARF: Any-angle routing for flow-based microfluidic biochips." In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 37.12 (2018), pp. 3042–3055.

[12] Q. Wang, H. Zou, H. Yao, T.-Y. Ho, R. Wille, and Y. Cai. "Physical co-design of flow and control layers for flow-based microfluidic biochips." In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 37.6 (2017), pp. 1157–1170.

[13] T.-M. Tseng, M. Li, B. Li, T.-Y. Ho, and U. Schlichtmann. "Columba: Co-layout synthesis for continuous-flow microfluidic biochips." In: *Proceedings of the 53rd Annual Design Automation Conference*. 2016, pp. 1–6.

[14] M. Li, T.-M. Tseng, B. Li, T.-Y. Ho, and U. Schlichtmann. "Component-oriented high-level synthesis for continuous-flow microfluidics considering hybrid-scheduling." In: *Proceedings of the 54th Annual Design Automation Conference 2017*. 2017, pp. 1–6.

[15] B. Crites, R. Sanka, J. Lippai, J. McDaniel, P. Brisk, and D. Densmore. "ParchMint: a microfluidics benchmark suite." In: *2018 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE. 2018, pp. 78–79.

[16] R. Sanka, J. Lippai, D. Samarasekera, S. Nemsick, and D. Densmore. "3DµF—interactive design environment for continuous flow microfluidic devices." In: vol. 9. 2019, p. 9166.

[17] M. Roderick, J. MacGlashan, and S. Tellex. "Implementing the deep q-network." In: *arXiv preprint arXiv:1711.07478* (2017).

[18] J. Schulman, S. Levine, P. Abbeel, M. Jordan, and P. Moritz. "Trust region policy optimization." In: *International conference on machine learning*. PMLR. 2015, pp. 1889–1897.

[19] V. Konda and J. Tsitsiklis. "Actor-critic algorithms." In: *Advances in neural information processing systems* 12 (1999).

[20] V. Mnih, A. P. Badia, M. Mirza, A. Graves, T. Lillicrap, T. Harley, D. Silver, and K. Kavukcuoglu. "Asynchronous methods for deep reinforcement learning." In: *International conference on machine learning*. PMLR. 2016, pp. 1928–1937.

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

[22] T. P. Lillicrap, J. J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver, and D. Wierstra. "Continuous control with deep reinforcement learning." In: *arXiv preprint arXiv:1509.02971* (2015).

[23] M. Hausknecht and P. Stone. "Deep reinforcement learning in parameterized action space." In: *arXiv preprint arXiv:1511.04143* (2015).

[24] W. Python. "Python." In: *Python Releases for Windows* 24 (2021).

[25] G. Van Rossum, F. L. Drake, et al. *Python reference manual*. Vol. 111. Centrum voor Wiskunde en Informatica Amsterdam, 1995.

[26] D. Kuhlman. *A python book: Beginning python, advanced python, and python exercises*. Dave Kuhlman Lutz, 2009.

[27] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, et al. "Pytorch: An imperative style, high-performance deep learning library." In: *Advances in neural information processing systems* 32 (2019).

[28] B. Pang, E. Nijkamp, and Y. N. Wu. "Deep learning with tensorflow: A review." In: *Journal of Educational and Behavioral Statistics* 45.2 (2020), pp. 227–248.

[29] T.-M. Tseng, M. Li, D. N. Freitas, T. McAuley, B. Li, T.-Y. Ho, I. E. Araci, and U. Schlichtmann. "Columba 2.0: a co-layout synthesis tool for continuous-flow microfluidic biochips." In: vol. 37. 8. 2017, pp. 1588–1601.

[30] L. P. Kaelbling, M. L. Littman, and A. W. Moore. "Reinforcement learning: A survey." In: *Journal of artificial intelligence research* 4 (1996), pp. 237–285.

[31] M. A. Wiering and M. Van Otterlo. "Reinforcement learning." In: *Adaptation, learning, and optimization* 12.3 (2012), p. 729.

[32] Y. Li. "Deep reinforcement learning: An overview." In: *arXiv preprint arXiv:1701.07274* (2017).

[33] D. Silver, G. Lever, N. Heess, T. Degris, D. Wierstra, and M. Riedmiller. "Deterministic policy gradient algorithms." In: *International conference on machine learning*. Pmlr. 2014, pp. 387–395.

[34] S. Wang. *Deterministic Policy Gradient (DPG) A slide for illustration of DPG principles, page 4*. 2023. URL: `https://github.com/wangshusen/DRL/blob/master/Slides/6_Continuous_2.pdf`.

[35] Y. Hou, L. Liu, Q. Wei, X. Xu, and C. Chen. "A novel DDPG method with prioritized experience replay." In: *2017 IEEE international conference on systems, man, and cybernetics (SMC)*. IEEE. 2017, pp. 316–321.

[36] J. Suárez-Varela, A. Mestres, J. Yu, L. Kuang, H. Feng, P. Barlet-Ros, and A. Cabellos-Aparicio. "Feature engineering for deep reinforcement learning based routing." In: *ICC 2019-2019 IEEE International Conference on Communications (ICC)*. IEEE. 2019, pp. 1–6.

[37] U. Khurana, H. Samulowitz, and D. Turaga. "Feature engineering for predictive modeling using reinforcement learning." In: *Proceedings of the AAAI Conference on Artificial Intelligence*. Vol. 32. 1. 2018.

[38] R. Bro and A. K. Smilde. "Principal component analysis." In: *Analytical methods* 6.9 (2014), pp. 2812–2831.

[39] M. D. Zeiler. "Adadelta: an adaptive learning rate method." In: *arXiv preprint arXiv:1212.5701* (2012).

[40] L. Buşoniu, D. Ernst, B. De Schutter, and R. Babuška. "Online least-squares policy iteration for reinforcement learning control." In: *Proceedings of the 2010 American Control Conference*. IEEE. 2010, pp. 486–491.

[41] X. Li, Y. Ma, and C. Belta. "A policy search method for temporal logic specified reinforcement learning tasks." In: *2018 Annual American Control Conference (ACC)*. IEEE. 2018, pp. 240–245.

[42] S.-C. Lin, I. F. Akyildiz, P. Wang, and M. Luo. "QoS-aware adaptive routing in multi-layer hierarchical software defined networks: A reinforcement learning approach." In: *2016 IEEE International Conference on Services Computing (SCC)*. IEEE. 2016, pp. 25–33.

[43] B. Baker, O. Gupta, N. Naik, and R. Raskar. "Designing neural network architectures using reinforcement learning." In: *arXiv preprint arXiv:1611.02167* (2016).

[44] T. Salimans and D. P. Kingma. "Weight normalization: A simple reparameterization to accelerate training of deep neural networks." In: *Advances in neural information processing systems* 29 (2016).

[45] A. Botchkarev. "A new typology design of performance metrics to measure errors in machine learning regression algorithms." In: *Interdisciplinary Journal of Information, Knowledge, and Management* 14 (2019), pp. 045–076.

[46] L. Rosenfeld, T. Lin, R. Derda, and S. K. Tang. "Review and analysis of performance metrics of droplet microfluidics systems." In: *Microfluidics and nanofluidics* 16.5 (2014), pp. 921–939.

[47] K.-H. Tseng, S.-C. You, W. H. Minhass, T.-Y. Ho, and P. Pop. "A network-flow based valve-switching aware binding algorithm for flow-based microfluidic biochips." In: *2013 18th Asia and South Pacific Design Automation Conference (ASP-DAC)*. IEEE. 2013, pp. 213–218.

[48] D. R. Mott, P. B. Howell Jr, J. P. Golden, C. R. Kaplan, F. S. Ligler, and E. S. Oran. "Toolbox for the design of optimized microfluidic components." In: *Lab on a Chip* 6.4 (2006), pp. 540–549.

[49] R. Derakhshan, A. Mahboubidoust, and A. Ramiar. "Design of a novel optimized microfluidic channel for CTCs separation utilizing a combination of TSAWs and DEP methods." In: *Chemical Engineering and Processing-Process Intensification* 167 (2021), p. 108544.

[50] J. D. Anderson and J. Wendt. *Computational fluid dynamics*. Vol. 206. Springer, 1995.

[51] T. J. Chung. *Computational fluid dynamics*. Cambridge university press, 2002.

[52] J. F. Wendt. *Computational fluid dynamics: an introduction*. Springer Science & Business Media, 2008.