SANDIA REPORT
SAND2022-13074
Printed September 2022

Sandia National Laboratories

AI-Enhanced Codesign for Next-Generation Neuromorphic Circuits and Systems

Douglas Cale Crowder, J. Darby Smith, and Suma George Cardwell

Cognitive and Emerging Computing
Center for Computing Research
Sandia National Laboratories
Albuquerque, NM, 87185

{dccrowd, jsmit16, sgcardw}@sandia.gov

Prepared by
Sandia National Laboratories
Albuquerque, New Mexico 87185
Livermore, California 94550

Issued by Sandia National Laboratories, operated for the United States Department of Energy by National Technology & Engineering Solutions of Sandia, LLC.

Available to DOE and DOE contractors from

U.S. Department of Energy
Office of Scientific and Technical Information
P.O. Box 62
Oak Ridge, TN 37831

Telephone: (865) 576-8401
Facsimile: (865) 576-5728
E-Mail: reports@osti.gov
Online ordering: http://www.osti.gov/scitech

Available to the public from

U.S. Department of Commerce
National Technical Information Service
5301 Shawnee Road
Alexandria, VA 22312

Telephone: (800) 553-6847
Facsimile: (703) 605-6900
E-Mail: orders@ntis.gov
Online order: https://classic.ntis.gov/help/order-methods

ABSTRACT

This report details work that was completed to address the Fiscal Year 2022 Advanced Science and Technology (AS&T) Laboratory Directed Research and Development (LDRD) call for "AI-enhanced Co-Design of Next Generation Microelectronics." This project required concurrent contributions from the fields of 1) materials science, 2) devices and circuits, 3) physics of computing, and 4) algorithms and system architectures.

During this project, we developed AI-enhanced circuit design methods that relied on reinforcement learning and evolutionary algorithms. The AI-enhanced design methods were tested on neuromorphic circuit design problems that have real-world applications related to Sandia's mission needs. The developed methods enable the design of circuits, including circuits that are built from emerging devices, and they were also extended to enable novel device discovery. We expect that these AI-enhanced design methods will accelerate progress towards developing next-generation, high-performance neuromorphic computing systems.

ACKNOWLEDGMENT

The authors acknowledge the support of Frances Chance, Suhas Kumar, Paul Kuberry, and our program manager Mike Descour for their invaluable input and feedback during this project.

The authors acknowledge support from Advanced Science and Technology (AS&T) Laboratory Directed Research and Development (LDRD) program. Sandia National Laboratories is a multi-mission laboratory managed and operated by National Technology and Engineering Solutions of Sandia, LLC, a wholly owned subsidiary of Honeywell International, Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525. This report describes technical results and analysis. Any subjective views or opinions that might be expressed in the paper do not necessarily represent the views of the U.S. Department of Energy or the United States Government.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

SUMMARY

In this work, we began to develop a codesign framework for circuit design (MAMMAL- Modular and Multi-level Machine Learning) with a focus on developing next-generation neuromorphic components, circuits, and systems. Neuromorphic computing is a field of active research with many competing approaches in digital, analog and mixed-signal. Leading neuromorphic platforms (e.g. Loihi) rely upon compact neuron models that lend themselves well to extreme scalability. However, as seen in nature (e.g. Dragonfly TSDNs - target selective descending neurons), fewer complex neurons can implement complex computations. We specifically evaluate the dendrite analog circuit as an example problem. Dendrites are highly branched tree-like structures that connect a neuron's synapses to the soma. Research shows that dendrites can perform operations such as non-linear filtering, spatial and temporal summation of synaptic inputs, coincidence detection, synaptic scaling, and sequence detection [26, 21]. Biological dendrites are also known for their complex physical structures with significantly greater fan-in ($\approx$ 10,000 inputs). Our work will enable exploration of a complex neuron model with dendritic connections. Analog neuromorphic approaches offer savings in energy compared to digital neuromorphic, with the promise of more complexity and dynamics. Previous work has demonstrated dendritic models using sub-threshold analog floating-gate transistors [13, 35]. The voltage across these devices has been shown to obey a class of non-linear PDEs. Our hypothesis is that the technology to implement nonlinear summation currently exists, but capturing this capability will require specific configuration of the neural network. It was therefore critical that the architecture, hardware, and algorithm development be conducted in tight synergistic collaboration for this co-design initiative. MAMMAL can enable this.

We utilized reinforcement learning techniques as the driving mechanism for codesign. This AI/ML approach is constrained by input/output characteristics, device physics, and circuit topology. Using these constrains, reinforcement learning is able to design circuits. The outcome in this scenario is a desired neural circuit function, including any SWaP constraints. In the process of creating this strategy, different circuit components drop out or are rearranged in unexpected and revolutionary ways. In this way, AI/ML enable the combination of algorithms, hardware, and physics for the discovery and codesign of new neural circuits.

We used the Passive Dendrite cable model as an example to test our AI-enhanced tool. Experiments conducted included: 1) delay lines, 2) delay and detect circuits for pattern detection and 3) dendrite-inspired delay and detect circuits that used an abstracted model for the dendrite sub-circuit. We demonstrated "learned" dendritic circuits leveraging abstracted device models for pattern recognition. We also studied the differences in reinforcement learning (RL) and evolutionary algorithms (EA) for a given circuit. EA can be used for rapid prototyping with novel modules and more creativity, and RL can be used for 'production' circuit design that involves designing many different circuits from the same modules. By combining EA and RL methods, it

may be possible to rapidly develop creative designs. We used curriculum learning to accelerate training for larger circuits. We also demonstrated examples leveraging an emerging device like the Mott Memristor [23] and evolutionary algorithms for circuit design.

We also explored novel device discovery using reinforcement learning. Using reinforcement learning, we were able to build circuits from novel device models that were optimized using AI. We show that, by analyzing the models of novel devices, we can gain insight into the types of novel devices that we should be trying to develop. And, we also show that imperfect manifestations of the novel device models can still be used by RL to solve known problems.

NOMENCLATURE

Table 0-1. Nomenclature

Abbreviation	Definition
AI	Artificial Intelligence
ASIC	Application Specific Integrated Circuit
AS&T	Advanced Science and Technology
DOE	Department of Energy
EA	Evolutionary Algorithms
EDA	Electronic Design Automation
EONS	Evolutionary Optimization of Neuromorphic Systems
IC	Integrated Circuit
ML	Machine Learning
MOSFET	Metal-Oxide-Semiconductor Field-Effect Transistor
ND	Nuclear Deterrence
NSP	National Security Program
PDE	Partial Differential Equations
RL	Reinforcement Learning
SNR	Signal-to-Noise Ratio
TPU	Tensor Processing Unit
VLSI	Very Large Scale Integration

1. INTRODUCTION

1.1. Motivation

Circuit design is a labor-intensive process that must be performed by highly-trained professionals who have years of experience. In niche circuit design fields, subject matter experts require additional training, which further limits the available talent. Because circuit design talent is limited, mission-critical circuit design knowledge is at risk of being lost due to normal employee turnover.

In order to maintain its technological dominance and national security, the United States needs to augment its circuit design workforce. We propose to augment the United States circuit design workforce by developing AI-enhanced tools that can automate many circuit design tasks and also efficiently encode domain-knowledge.

1.2. AI-Enhanced Microelectronics Design

Many works have explored the possibility of using AI-enhanced tools to optimize existing microelectronics designs. While many of these works have focused on optimizing human designs, some works have explore the possibility of optimizing random initial designs [45]. Existing AI-enhanced tools have been used to optimize device placement [43, 42, 19], connection routing [42, 19], component sizing [6], digital logic design [58, 43, 42, 19], software to hardware mapping [31, 56], SPICE simulation [42], lithography [42, 19], device modeling [42], runtime management [42], and high-level synthesis [19]. Several AI-enhanced electronic design automation (EDA) tools are even commercially available [50, 7]. Generally, the workflow for AI-enhanced optimization involves the following steps:

1. A human produces an initial design
2. The initial design is transformed into a computer-interpretable representation (commonly, a graph or a graph neural network)
3. AI is used to optimize the human design

Existing AI-enhanced optimization techniques have already produced super-human designs in super-human time [32]. Here, we address the rate-limiting step of AI-enhanced microelectronics optimization: the production of the initial human design.

In this work, we detail an AI-enhanced circuit design tool that is capable of automating circuit design for simple neuromorphic circuits. Unlike previous works, our AI-enhanced circuit design tools do not require an initial human design; they can design circuits “from scratch.”

1.3. AI-Enhanced Microelectronics Discovery

During a top-down design process, an AI-enhanced circuit design tool chooses components to create a circuit that solves an interesting problem. During a bottom-up design process, an AI-enhanced circuit design tool starts with (potentially novel) components and explores the problems that can be solved with the provided components. Both design processes start with existing components and problems.

Suppose that a problem cannot be solved with any combination of known components. Under these circumstances, an AI-enhanced circuit design tool will need to do more than use known components - the tool will need to discover a new component. Here, we present AI-enhanced methods for stimulating microelectronic discovery. Specifically, we present methods that allow AI-enhanced tools to suggest specifications for novel components that, if physically realized, can be used to solve known problems.

1.4. Hardware Accelerators and Neuromorphic Architectures

Recent successes in AI have been facilitated not only by algorithmic innovations, but also by improvements in hardware [17]. With the rapid adoption of AI in many different fields, the demand for specialized computation is growing. Increasingly, companies like Google and Apple are developing their own integrated circuits (ICs). The industry is shifting towards more application specific integrated circuits (ASICs) to maximize computational efficiency and improve software-hardware integration [47]. There has been an explosion of machine learning accelerator approaches like Google's Tensor Processing Unit (TPU), Cerebras [24] (wafer-scale), and Mythic (analog) [1], each with a unique approach to overcoming performance bottlenecks.

Neuromorphic architectures are another example of hardware accelerators that are expected to improve computing in the next decade and beyond as non-Von Neumann architectures gain prominence. Implementations of neuromorphic chips in silicon exist today (e.g. Loihi, TrueNorth, SpiNNaker, BrainChip, GrAI Matter Labs). Non-CMOS approaches are promising, and industry trends (Imec/Global Foundries) show that these architectures will be available for mass production soon. Neuromorphic accelerators can impact the efficiency of machine learning, scientific computing, and edge applications with 2-3 orders of magnitude improvement in energy and speed.

Neuromorphic computing is expected to improve power efficiency scaling and meet the DOE's future mission demands for high performance computing (HPC) (nuclear deterrence (ND), climate simulations) and edge computing (satellite, space systems, national security programs (NSPs), global security). The Department of Energy (DOE) Basic Research Needs for Microelectronics report has emphasized the need for codesigned devices, circuits, algorithms, and architectures. AI-enhanced codesign of next-generation microelectronics will enable algorithm and hardware innovations that will accelerate progress towards next-generation high-performance neuromorphic computing systems.

1.5. Neuromorphic Computing

Neuromorphic computing mimics key biological properties of brains in silicon. The building blocks in a neuromorphic architecture consist of neurons, synapses, dendrites, routing etc. that are modeled using devices and circuits. The field of neuromorphic computing was pioneered by Carver Mead at CalTech in late 80s [29]. Initial circuits developed were analog circuits that leveraged the sub-threshold dynamics of CMOS transistors to emulate biological components. Many neuromorphic chips have been developed to date including digital [10, 30, 38, 18, 11], analog [5, 41, 14], mixed-signal [34, 39, 4] and beyond-CMOS [53] neuromorphic systems. There has also been a lot of work demonstrating the efficacy of beyond-CMOS devices as candidates to model neurons and synapses [23, 25, 55]. Digital system like Loihi and SpiNNaker have been scaled to 100 million [36] and billion neurons [28] respectively. Beyond-CMOS devices also promise scaling gains as well as computational efficiency. Leading neuromorphic platforms rely upon compact and simple neuron models for scaling. However, in order to achieve brain-like cognition, we need complexity (increased computational power per neuron, dendrites as computational interconnects, online learning) in addition to scaling (increased number of neurons and synapses).

Next-Generation Heterogeneous Neuromorphic Architectures: There are many on-going efforts to look at emerging devices for neuromorphic computing [23, 25, 55]. While we need novel neuromorphic devices to accelerate computation, we also need novel algorithms and architectures. We hypothesize that designing complex neurons will augment the capabilities that current neuromorphic systems offer. For example, introducing dendritic processing will introduce non-linear summation, spatio-temporal processing, and increased connectivity. Synaptic stochasticity can be leveraged to do probabilistic computation. Brain-inspired local learning is another area of active research that could impact the use of neuromorphic processors as not just inference engines, but also training engines.

Next-generation neuromorphic circuits and systems based upon nonlinear dendritic processing and local learning will balance the trade-off between scalability and the biological complexity. Novel approaches in fabrication like three-dimensional architectures, wafer-scale technology, and in-memory computing devices could further alleviate current communication and connectivity bottlenecks. This would require tools that enable synergistic collaboration across devices, architectures, software, and algorithms. An AI based approach to designing microelectronics could alleviate some of the challenges posed by full-stack design and effectively incorporate constraints posed by algorithms, architectures, circuits and devices.

The remainder of this chapter provides the necessary background for understanding our approach to AI-enhanced circuit design. In Section 1.6, we outline three potential approaches to AI-enhanced circuit design. In Section 1.7, we detail the needs and building blocks for constructing neuromorphic circuits, along with the goals and science questions for this project. Finally, in Section 1.8, we detail the remaining organization of this report.

1.6. AI Approaches to Circuit Design

In order to automate circuit design, we used reinforcement learning and evolutionary algorithms. Here, we provide details about these AI methods as well as one alternative method that we have yet to implement: game theory.

1.6.1. Game Theory

Game theory is a mathematical subject used to analyze strategies and interactions among competing or cooperating groups [52]. Traditionally, a game, in the game theoretic sense, consists of

- players;
- actions for players to take;
- strategies to determine how players choose actions;
- and payoffs determined by action choice.

The basic notion is that players select a strategy to play. This selection could be done in tandem with other players or in sequence, and other players may or may not know strategies selected by others. A game is played with players playing actions according to their strategy, and then rewards are given. Games may be iterative with the objective to find the best strategy for actions. The "best strategy" is typically a Nash equilibrium. A Nash equilibrium is a set consisting of a strategy for each player where no single player has anything to gain by deviating to a different strategy. In multi-player games this means you will not do better on average by switching to a different strategy. In two player zero-sum games (games where one player's gain is equivalent to another player's loss) this means neither player will lose out on average when both are playing the Nash equilibrium strategy.

Calculation of these equilibrium strategies is not trivial. Nonetheless, such strategies are obviously valuable. Game theory has been used to optimize strategies in a variety of fields, including business, economics, and transportation [57, 8, 15].

Broadening how one might view a player, game theory can be extended to system design. This can find applications when one can formulate a game between an adversary seeking to disrupt a design and a player seeking to protect by careful design. Indeed, such construction has proven to be useful in network security and network design [59]. Cooperative game theory, however, can be used to great utility in design with recent applications in optimal wireless network design [27]. Further cooperative modes can describe the interaction among designers that each produce a single piece of a larger system [51].

Previous successful application of game theory to system design suggests that game theory would also be useful for circuit design. Since circuits can be viewed as systems of devices that are connected in defined ways, and circuits can be considered sub-systems of larger systems, it would be useful to develop game theory methods that enable systems of systems design. To date, game theory has primarily been used to either model and analyze a system of systems or to optimize

control strategies, but game theory has not been used to design systems of systems from scratch [9, 3]. There is at least one explicit example of multi-objective circuit design optimization using game theory [12], but this is optimization rather than design.

A novel approach to linear circuit design using game theory could arise by constructing a cooperative game among players who place a single type of device or circuit component. Players take turns electing to play their piece or to not play their piece. All knowledge of previously played components is available to all players. Any player may choose to terminate the circuit instead of playing their component. Once a circuit is terminated, it is scored, and a payoff is given to all players if the circuit meets the desired expectations. Penalties may also be given out if certain characteristics of the circuits are not desirable (i.e., it meets expectations but would not hold up under certain temperatures).

One may ask: "Why are multiple players needed instead of just a single player that allocates all types of devices or components?" A single player game with slightly modified rewards resembles reinforcement learning (RL), which we discuss in the next section.

1.6.2. Reinforcement Learning

In reinforcement learning [49], an *agent* observes an *environment* (e.g., a game or system) and uses the *observation* to choose *actions*. The actions get applied to the environment, causing the environment to probabilistically transition to a new *state* and emit a *reward*. A *parameter update function* uses observations, actions, and rewards to train the agent to choose actions that maximize the expected sum of future rewards. Figure 1-1 provides an illustration of RL. A full description of the reinforcement learning framework is provided in Section 2.2.

In the context of circuit design, an RL agent is a circuit designer. The agent observes some representation of the circuit (e.g., the output of the circuit, given some input). Actions include placing components into the circuit and tuning component parameters. The RL agent is rewarded base on the performance of the circuit, given some pre-define metric of performance.

1.6.3. Evolutionary Algorithms

Evolutionary algorithms (EA) were inspired by the biological process of evolution. EA (Figure 1-2) starts by selecting a random initial population of *parents* that have a set of traits. In the context of circuit design, these traits might include the identity of components in the circuit, as well as the tunable parameters of the components. Parents are *selected* based on their *fitness* score, which is analogous to the sum of future rewards in RL. Selected parents then undergo a process of *crossover*, where offspring are produced by combining traits from selected parents. After crossover, children undergo *mutation*, where some traits are randomly modified. Children then become the new parents, and the process repeats.

Evolutionary algorithms have already been used to optimize neuromorphic circuits. Evolutionary Optimization of Neuromorphic Circuits (EONS) [45], for instance, optimizes networks of neuromorphic elements by considering the structure of the networks to be the parent traits.

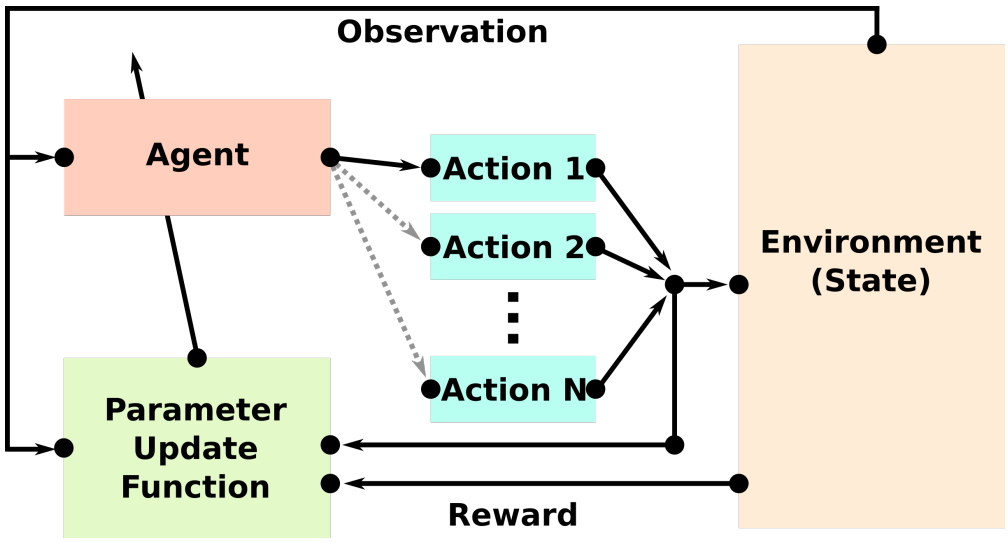


Figure 1-1. Overview of reinforcement learning. An *agent* observes an *environment* (e.g., a game or system) and uses the *observation* to choose *actions*. The actions get applied to the environment, causing the environment to probabilistically transition to a new *state* and emit a *reward*. A *parameter update function* uses observations, actions and rewards to train the agent to choose actions that maximize the expected sum of future rewards.

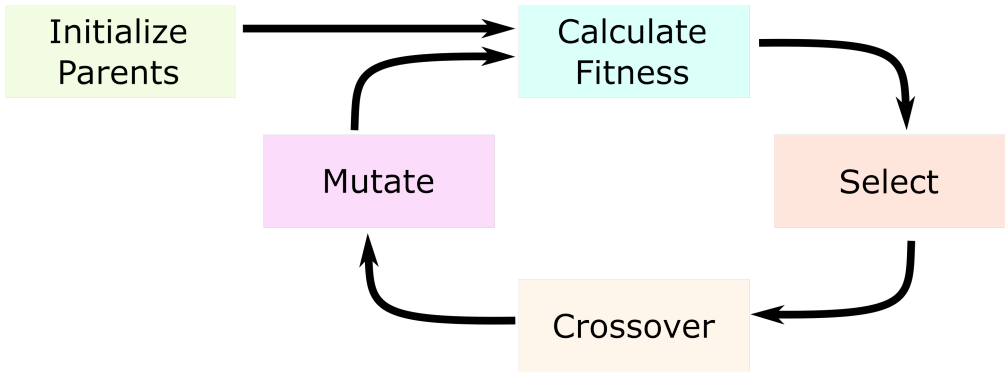


Figure 1-2. Overview of evolutionary algorithms. An initial cohort of parents is evaluated and then selected based on their fitness scores. Traits from selected parents are combined in the process of crossover, and then traits are randomly mutated. The resulting children become the new parents, and the process repeats.

Mutation functions change the networks by changing network connectivity or changing the identity of nodes in the network.

1.7. Building Neuromorphic Circuits using Reinforcement Learning

Innovation in circuits, especially those leveraging novel and emerging devices, will be crucial for building next-generation neuromorphic architectures. Automating circuit design will further accelerate design. However, this is still an active area of research. There are few instances of leveraging a game theoretic approach for circuit design optimization [12] and hardware (RTL) verification and security [48]. Lately, reinforcement techniques have been applied for placement and optimization [31, 32], generating layout for analog circuits [46] and optimizing analog circuits [54].

Key science questions we wanted this project to answer were as follows:

- Can we use techniques such as reinforcement learning for circuit design? What are the challenges associated with doing so?
- Will developing AI-enhanced codesign tools for neuromorphic circuits and architectures accelerate design for next-generation neuromorphic architectures?
- Will leveraging an AI-accelerated approach enable fast design of dendritic neural circuits? Dendritic neural circuits can be used as computational interconnects in a neuromorphic architecture.
- How would our efforts impact neuromorphic accelerators with applications in machine learning, scientific computing, and brain-like simulation with significant improvement in energy and speed?

Our approach also looks to answer these questions and more. We envision designing an AI/ML approach constrained by domain knowledge, device physics, SWaP (Size, Weight and Power), and circuit topology. For example, RL can be used to develop a policy network which enables designing different circuit topologies in unexpected and revolutionary ways. In this way, AI enables the combination of algorithms, hardware, and physics of the devices for the discovery and codesign of novel neural circuits.

1.8. Report Outline

In Chapter 2, we describe our methods in detail. Specifically, we will describe the AI methods that we used as well as the tasks that we used to compare different AI methods. In Chapter 3, we detail the experiments that we used to guide the development of AI-enhanced circuit design tools. Specifically, we describe how AI-enhanced circuit design tools evolved from "delay line" tasks to signal discrimination tasks. We also discuss curriculum learning and transfer learning methods for accelerating AI-enhanced circuit design. In Chapter 4, we compare RL and evolutionary algorithms on the same design task, and we demonstrate that RL has advantages as a production

tool, but evolutionary algorithms have advantages for rapid prototyping. In Chapter 5, we demonstrate that AI-enhanced circuit design methods can be used to rapidly prototype circuits that are built from emerging devices, such as the Mott memristor. In Chapter 6, we move beyond AI-enhanced circuit design into AI-enhanced novel device discovery. We expect that the landmark results in this chapter will cause a paradigm shift in how we approach discovery in the field of microelectronics in the future. Finally, we conclude this report with Chapter 7 by detailing the impact of our AI circuit and device design discoveries, listing our innovations, and outlining future work.

2. METHODS FOR RL-BASED CIRCUIT DESIGN

In this chapter, we explain RL methods that were widely use throughout this project.

2.1. Tasks

Based on our project goals (see Section 1.7), we wanted to ultimately design energy-efficient dendrite-like neuromorphic circuits that could successfully perform signal detection. While we ultimately tested our AI-enhanced circuit design tools on signal detection tasks (Section 2.1.2), we initially tested the tools on "delay line" tasks (Section 2.1.1).

2.1.1. Delay Line

Initially, we prototyped AI-enhanced circuit design methods on simple "delay line" circuits. We chose to use delay line circuits because they are the simplest circuits that capture interesting properties of neuromorphic circuits: 1) components are capable of accepting inputs and 2) there is a time delay associated with each processing step.

As shown in Figure 2-1, the purpose of a delay line circuit is to delay an input X by a set amount of time. For simple neuromorphic circuits, this input might be a spike representing a zero or a one. To build a delay line circuit, it was necessary to choose the appropriate neuromorphic components and tune their parameters.

For the initial delay line experiments, we used circuit design tools that were based on RL. We did not attempt to use evolutionary algorithms. At each time step, the RL agent observed 2 numbers: 1) the desired delay and 2) the total delay already in the circuit. The RL agent needed to choose to

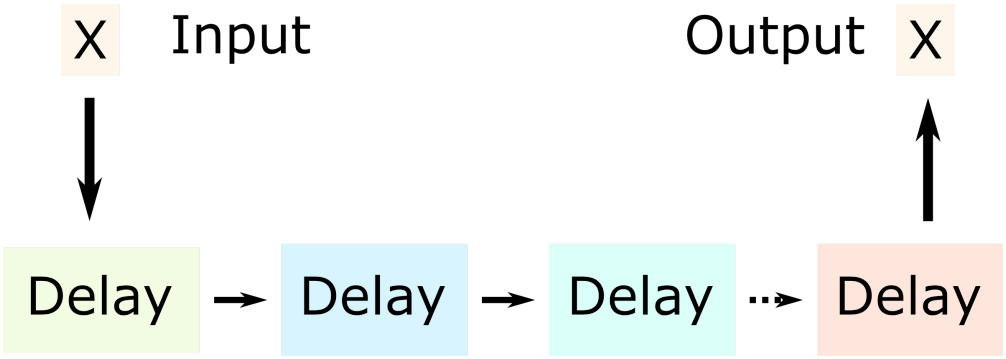


Figure 2-1. Example of a delay line. An AI-enhanced circuit design tool must place delay components to delay the input X by the correct amount.

Component Name	Function
Probe	Terminate the circuit
Delay1	Delay the input by 1 time step
Delay10	Delay the input by 10 time steps
Delta Delay	Delay the input by δ time steps. δ is tunable.

Table 2-1. Delay line components.

place one of several components, as detailed in Table 2-1. Component diagrams can be found in Appendix A. Some components had tunable parameters that the RL agent also needed to choose. Circuit design terminated when the RL agent placed a "Probe" component (the name refers to where you would place a probe to measure the circuit output).

RL agents were provided with sparse rewards (see Section 2.2.6) at the terminal state, only, based on the the absolute value of the difference $|\Delta|$ between the desired delay and the output delay. For initial experiments, which only involved Delay1 and Probe components, the reward r was given by:

$$r = -1 \times |\Delta| \tag{2.1}$$

which penalized the RL agent for producing a delay that was different than desired delay. For subsequent experiments, we modified the reward to be:

$$r = 1 - \frac{|\Delta|}{2} \tag{2.2}$$

This reward was designed to produce occasional positive rewards, which can allow RL to converge faster by preventing unnecessary exploration of the solution space. We then further modified the reward function to penalize positive (i.e. too much) delay:

$$r = \begin{cases} (1 - \frac{|\Delta|}{2}) \times 0.95, & \text{if } \Delta > 0 \\ (1 - \frac{|\Delta|}{2}), & \text{otherwise} \end{cases} \tag{2.3}$$

This reward was designed to accelerate convergence by encouraging RL to not make decisions that it could not reverse. With this reward, the RL learned to "fix" negative delays by adding additional delay components and also learned not to make unfixable mistakes (i.e. positive delays). We then log-transformed Δ to ensure that the RL agent did not ignore relatively small delays.

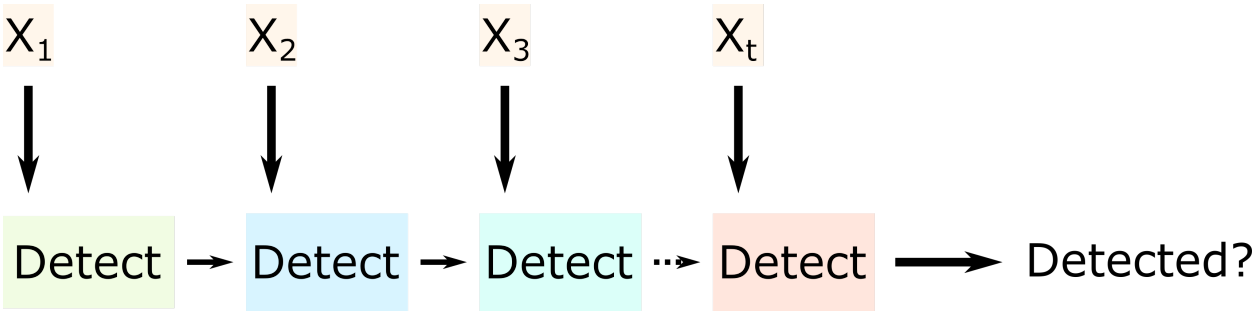


Figure 2-2. Example of a detect circuit. An AI-enhanced circuit design tool must place detect components that detect if each of t samples of a signal, X, is the part of the target pattern.

$$r = \begin{cases} (1 - \frac{|\log_{10}(\Delta)|}{2}) \times 0.95, & \text{if } \Delta > 0 \\ (1 - \frac{|\log_{10}(\Delta)|}{2}), & \text{otherwise} \end{cases} \tag{2.4}$$

In some experiments, we tested if RL agents could maximize performance while minimizing component costs. To do this, we assigned cost c_t to the t^{th} of T components and updated the reward function to:

$$r = \begin{cases} (1 - \frac{|\log_{10}(\Delta)|}{2} - \sum_0^T c_t) \times 0.95, & \text{if } \Delta > 0 \\ (1 - \frac{|\log_{10}(\Delta)|}{2} - \sum_0^T c_t), & \text{otherwise} \end{cases} \tag{2.5}$$

2.1.2. Signal Detection

After prototyping AI-enhanced circuit design methods on delay line circuits, we increased the task difficulty by requiring the circuits to detect if an input pattern matched a target pattern. As shown in Figure 2-2, the t^{th} component of every circuit was responsible for detecting whether the t^{th} sample of a signal, X, matched the t^{th} sample of a target pattern.

For the initial signal detection experiments, we used circuit design tools that were based on RL. We did not attempt to use evolutionary algorithms. At each time step, the RL agent observed 16 examples of a target signal and 16 examples of non-target signals (Figure 2-3).

Target signals and non-target signals were drawn from several distributions to test different capabilities of the RL agents (Figure 2-4). The target and non-target distributions had matching distributions at some subset of the samples. Typically, 60% of the samples were drawn from different distributions for the target and non-target distributions. During transfer learning and curriculum learning (see Section 2.2.7), the total number of differing samples remained constant (meaning that the proportion of differing samples decreased).

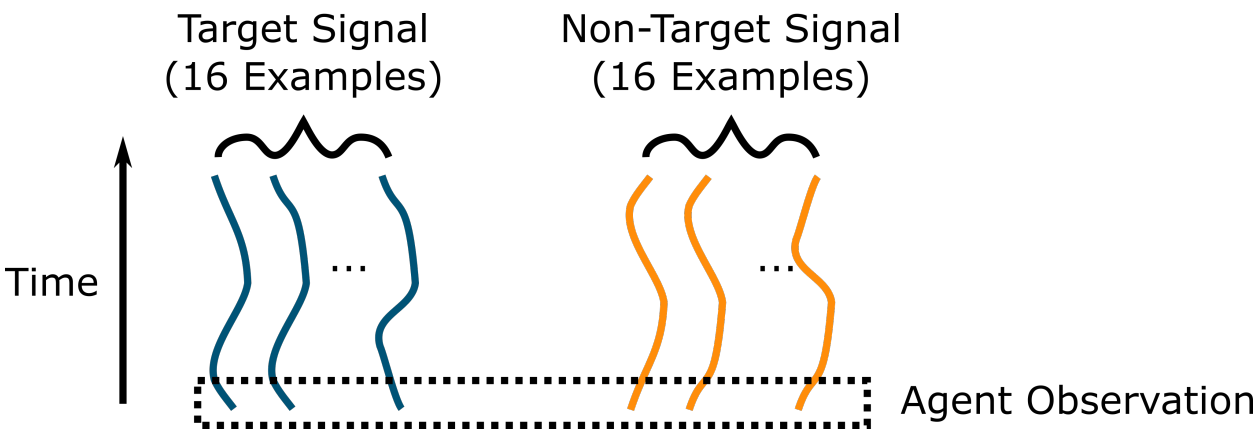


Figure 2-3. Observing the target signal. At each timestep, the RL agent observed 16 examples of the target signal and 16 examples of non-target signals. The agent observation "moves up" the signals as time passes. At each time step, the agent observes one sample from each signal.

The parameters of the target and non-target signal distributions were changed to modulate the signal-to-noise ratio (SNR) between the target signal (signal) and non-target signal (noise). See Figure 2-4 for illustrations. In the "Digital" condition, examples of target and non-target signals were drawn from $U\{0,1\}$ and then approximately 60% of samples for the target signal were forced to be equal. In the "Low SNR" condition, all samples of the non-target signal were drawn from a uniform distribution $U(-1,1)$; the samples of the target signal were drawn from a uniform distribution $U(l_t,h_t)$ where $h_t = l_t + 0.2$ and $l_t \sim U(-1,0.8)$. In the "Medium SNR" condition, all samples of the non-target signal were drawn from a uniform distribution $U(-1,1)$; the samples of the target signal were drawn from a uniform distribution $U(l_t,h_t)$ where $h_t = l_t + 0.2$, $|l_t| \sim U(0.4,0.8)$, and $\text{sign}(l_t) \sim U\{-1,1\}$. Put simply, the Medium SNR condition increased the SNR by forcing the target signal to have a larger magnitude. In the "High SNR" condition, all samples of the non-target signal were drawn from a uniform distribution $U(-0.1,0.1)$; the samples of the target signal were drawn from a uniform distribution $U(l_t,h_t)$ where $h_t = l_t + 0.2$, $|l_t| \sim U(0.4,0.8)$, and $\text{sign}(l_t) \sim U\{-1,1\}$. Put simply, the High SNR condition increased the SNR by decreasing the magnitude of the non-target signal by a factor of 10. In the "Separable About Zero" condition, the magnitude of the non-target signal was drawn from $U(0,1)$; the samples of the target signal were drawn from a uniform distribution $U(l_t,h_t)$ where $h_t = l_t + 0.2$, $|l_t| \sim U(0.4,0.8)$, and $\text{sign}(l_t) \sim U\{-1,1\}$, and the signs of the target and non-target distributions were constrained to be opposite. The purpose of the Separable About Zero condition was to create a signal discrimination task that was easy for a human to solve with perfect accuracy by placing components that act as positive number detectors and negative number detectors. Because humans can solve this problem easily, it is easier to interpret solutions that are found by AI-enhanced circuit design tools.

To discriminate the target signal from non-target signals, the RL agent needed to choose to place one of several components, as detailed in Table 2-2. Component diagrams can be found in Appendix A. Some components had tunable parameters that the RL agent also needed to choose. Circuit design terminated when the RL agent placed a "Probe" component.

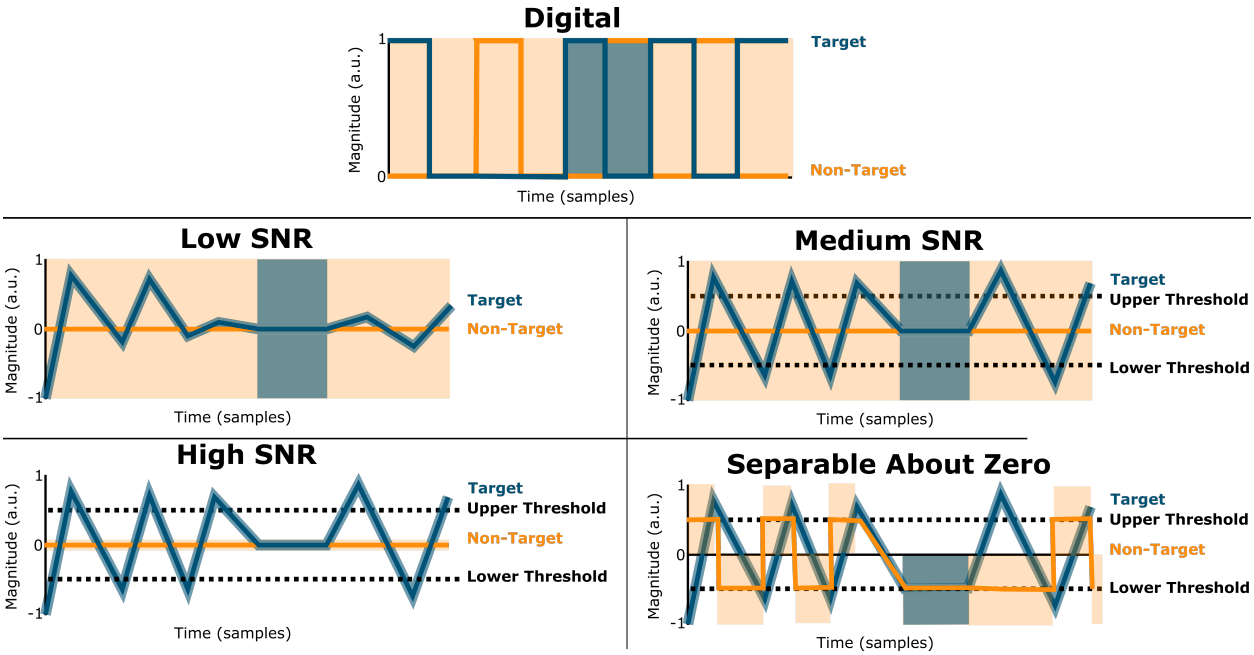


Figure 2-4. Target and non-target signal distributions. Bold lines are the mean (or mode for digital); shaded regions represent the range. All distributions were uniform across the range. Target and non-target distributions differed at a subset of samples - typically about 60%. In the Digital condition, approximately 60% of the time, all examples of the target signal matched, whereas the non-target samples were drawn from $U\{0,1\}$. In the Low SNR condition, the time-varying target distribution and the non-target distribution overlapped. In the Medium SNR condition, the mean of the target distribution was required to be greater in magnitude than some threshold, which was chosen to be ± 0.5. In the High SNR condition, the range of the non-target distribution was decreased by a factor of 10. In the Separable About Zero condition, the target and the non-target distributions always had separate signs. a.u. = "arbitrary units." SNR = "signal to noise ratio."

Component Name	Function
Probe	Terminate the circuit
Delay	Do not detect any number. The output of this component is the output of the previous component.
Detect0	Detect a 0
Detect1	Detect a 1
DetectAny	Detect a number that is $\mu \pm 0.11$, where μ is a tunable parameter.

Table 2-2. Delay line components.

For detection experiments, we tested both sparse and dense rewards (see Section 2.2.6). For the Digital signal detection task, the dense reward r_t was given by:

$$r_t = \begin{cases} 1, & \text{if correct component} \\ 0, & \text{otherwise} \end{cases} \tag{2.6}$$

For the Digital signal detection task, the sparse reward r_t was given by:

$$r = \begin{cases} 1, & \text{if circuit terminated \& correct circuit} \\ 0, & \text{otherwise} \end{cases} \tag{2.7}$$

For analog (non-Digital) detection task, only sparse rewards were given. Rewards were calculated per example (i.e., for each of the 16 examples of target signal and each of the 16 examples of the non-target signal), and then the total reward was calculated according to:

$$\frac{\#\text{ correct} - \#\text{ incorrect}}{\text{total }\#} \tag{2.8}$$

The maximum possible reward was 1 if all examples were correctly classified. The minimum possible reward was -1 if all signals were incorrectly classified. A single mis-classification would result in a reward of $(31-1)/32 = 0.9375$. We chose this reward formulation so that chance classification resulted in a reward of 0. Negative rewards promote exploration, which can lead to slow convergence, but may be useful for locating global optima. Positive rewards promote exploitation of existing knowledge, which can lead to faster convergence, but the solution may converge to a local optima. By allowing chance classification to have a reward of 0, we hoped to promote rapid convergence to a non-trivial solution. Specifically, we wanted to avoid convergence to trivial solutions that are local optima, such as building circuits that always classify signals as target signals.

2.1.3. Dendritic Detection

In Section 2.1.2, we described a signal detection task that used idealized components that had well-defined detection ranges. Here, we consider how we can perform the same tasks that were described in Section 2.1.2, but while building circuits using neuromorphic components that approximate the function of biological neurons. Specifically, we were interested in modeling sub-threshold (non-spiking) activity in biological dendrites.

The dendritic detection components are described in Table 2-3, and a general model of dendritic components is provided in Figure 2-5. Additional diagrams are available in Appendix A. The dendritic components that we explored all implemented modified leaky integration models. This

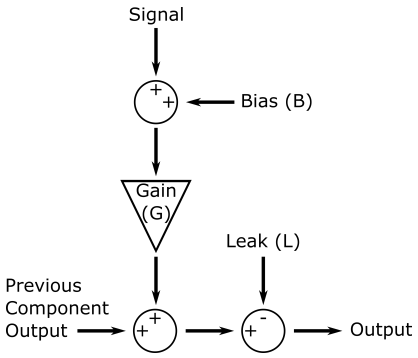


Figure 2-5. Diagram of dendritic component.

Component Name	Function
Delay	Do not detect any number. The output of this component is the output of the previous component.
Dendritic Detect	$G \in [-10, 10]$, $B = 0$, $L = 0$
Leaky Dendritic Detect	$G \in [-10, 10]$, $B = 0$, $L = 0.1$
Tunable Leaky Dendritic Detect	$G \in [-10, 10]$, $B = 0$, $L \in [-10, 10]$
Leaky Biased Dendritic Detect	$G \in [-10, 10]$, $B \in [-10, 10]$, $L = 0.1$

Table 2-3. Delay line components. G = gain. B = bias. L = Leak.

model approximates the way that current accumulates in dendritic compartments and spreads to other dendritic compartments. Given a signal sample X_t, an input from the previous component Y_{t-1}, a gain G, a bias B, and a leakage current L, the output of a dendritic component was given by:

$$Y_t = Y_{t+1} + G(X_t + B) - L \tag{2.9}$$

As described in Table 2-3, different components allowed G, B, and L to be tunable.

2.2. Reinforcement Learning

In Section 1.6.2, we provided an overview of reinforcement learning. Here, we provide details of the specific RL implementation that was used in this work.

2.2.1. Deep Q-Networks

For initial experiments, we used an RL algorithm known as Deep Q-networks (DQN) [33]. In DQN, a neural network is trained to predict the time-discounted expected sum of future rewards Q_t that will be received when in state s_t and taking action a_t. During evaluation or inference, the appropriate action is chosen by maximizing the expected sum of future rewards:

$$a_t = \arg\max_a Q_t(s_t, a) \tag{2.10}$$

During training, a hyperparameter ε determines the proportion of actions that are random. With probability ε, actions are chosen randomly; with probability $1 - \varepsilon$, rewards are chosen according to Equation 2.10. This randomness helps with exploration, which is essential to allow RL to converge to a good solution [49].

At each time step, the DQN-based RL agents observed a sample of the input signal (see Section 2.1.2 and Figure 2-4). The DQN-based RL agents used these observations to choose to place one of the components that were available in the library (see Tables 2-1, 2-2, and 2-3). DQN can only be used with discreet actions. Thus, it was not possible to use DQN with components that had real-valued tunable parameters.

2.2.2. Soft Actor-Critic

Because DQN can only produce discreet actions, we investigated soft actor-critic (SAC) [16], which is capable of producing continuous actions. SAC is a member of the actor-critic family of RL algorithms. The "actor" component of actor-critic algorithms refers to a neural network that observes the current state and predicts the action that will maximize the expected sum of time-discounted future rewards. The "critic" component of the actor-critic algorithm refers to a neural network that attempts to predict the sum of time-discounted future rewards that will be received when the agent is in state s_t and chooses a_t. The output of the critic is used by the parameter update function (see Figure 1-1) to update the actor.

Unlike DQN, SAC is capable of outputting real-valued actions. In fact, SAC outputs parameters for statistical distributions that describe these actions. The actual action is chosen by sampling from these statistical distributions. In SAC, these statistical distributions provide a source of exploration that helps the agents converge to good solutions.

Because SAC outputs real-valued actions, it is necessary to convert some of the SAC actions to discrete choices so that the (discreet) type of component can be chosen. SAC produces an action vector A after sampling from the statistical distributions. We chose the structure of A such that it was composed of 2 sub-vectors A_1 and A_2. Each component type that was represented in A_1 was also represented in A_2, but only if A_1 had a tunable parameter. The type of component to place was chosen by:

$$a_t = \arg\max_a A_1 \tag{2.11}$$

The real-valued tunable parameter for the component was the element of A_2 that corresponded to the chosen element in A_1.

2.2.3. Proximal Policy Optimization

Like SAC, proximal policy optimization (PPO) is a member of the actor-critic family of RL algorithms [44]. PPO is also capable of producing both discrete and real-valued actions using the same procedure that was described in Section 2.2.2. On-policy algorithms like PPO frequently converge in less clock time than off-policy algorithms like SAC. However, off-policy algorithms like SAC frequently converge with fewer simulation time steps. Because the neuromorphic circuit simulations were low-fidelity, and therefore time-efficient to run, we implemented PPO in later experiments to gain a clock-time advantage compared to SAC.

2.2.4. Hindsight Experience Replay

For delay experiments, the RL agent observed: 1) the desired delay and 2) the amount of delay already in the circuit. Because this observation space was so simple, it was possible to leverage hindsight experience replay (HER) [2] to accelerate learning. HER changes the target state retrospectively to allow the RL agents to gain more information during failure. For instance, suppose that an RL agent is asked to delay a signal by 5 timesteps, but the agent mistakenly delays the signal by 6 timesteps. HER allows the agent to say, "I didn't mean to delay by 6 timesteps, but if I need to delay by 6 timesteps in the future, I can use this set of actions."

HER can be used with either DQN or SAC. However HER is incompatible with proximal policy optimization [2].

2.2.5. Modular and Multi-Level Machine Learning (MAMMAL)

We were interested in taking the first steps towards true microelectronics codesign. True codesign requires:

1. System design across two or more system levels
2. Simultaneous or iterative top-down and bottom-up design processes

In this work, we created RL methods for performing top-down and bottom-up design. However, we were limited to design across a single system level (i.e., we designed circuits from components). We refer to the full AI-enhanced system codesign tool as Modular and Multi-Level Machine Learning (MAMMAL). As shown in Figure 2-6, MAMMAL accepts specifications from engineers about how a system should respond to a given set of inputs. MAMMAL also accepts design constraints. MAMMAL uses these specifications and design constraints to choose system components. In top-down mode, MAMMAL refines a system at progressively lower system levels. In bottom-up mode, MAMMAL works with a limited library of modules to try to solve problems. MAMMAL will also be capable of designing higher-level modules from lower-level modules. MAMMAL can already create specifications for novel modules if available modules in the library cannot be used to solve the problem of interest (see Chapter 6).

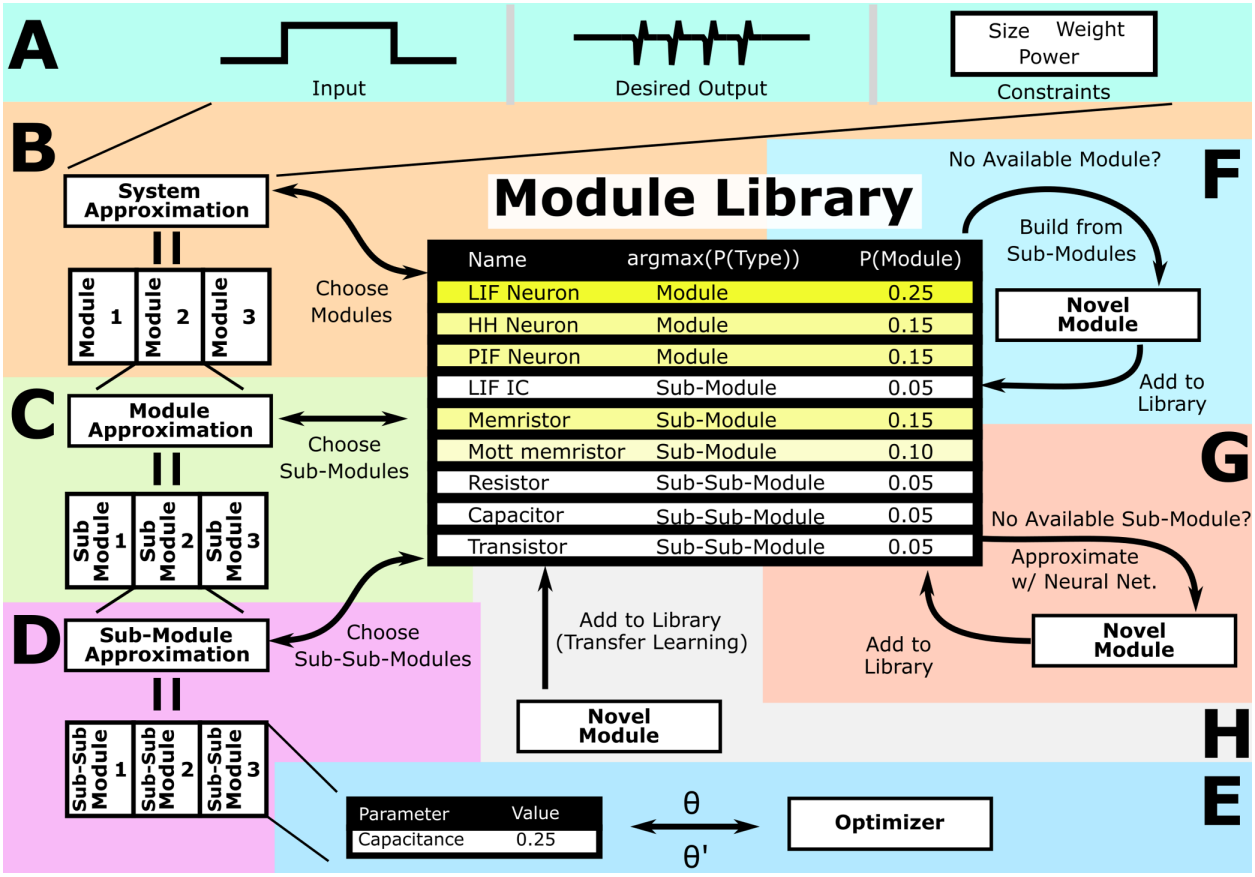


Figure 2-6. Modular and multi-level machine learning (MAMMAL). (A) A reinforcement learning agent is provided with system inputs, desired system outputs, and some constraints. (B) The RL creates a high-level approximation of the system using by choosing modules from a library. (C-D) The RL agent creates better approximations of the system by tuning submodules, which can also be chosen from the library. (E) Parameters for the lowest-level components are tuned. (F) If the RL agent is unable to create a high-level representation using existing modules, new modules can created from submodules. (G) If appropriate submodules cannot be identified, the RL agent can create specifications for novel modules that solve the problem of interest. (H) new modules can be added to the library using transfer learning. In this work, MAMMAL components A, B, E, and G were created. MAMMAL components C, D, F, and H still need to be created.

2.2.6. Rewards

We explored two basic reward strategies:

1. Dense rewards
2. Sparse rewards

Dense rewards are given to the RL agent after each component is placed. Essentially, after each component is placed, the RL agent is told if it placed the correct component or an incorrect component. Because dense rewards are given frequently, they contain a lot of information about how the RL agent can improve. This information can be exploited to help the RL agent learn more quickly.

Unfortunately, dense rewards can only be given if the evaluator knows what the "correct" design is. Of course, for real-world problems, this would mean that a human team would need to design each system before the RL designed the system; this, of course, is impractical. Thus, we explored sparse reward strategies.

Sparse reward strategies involve giving the RL agent rewards only at the end of the design process. These rewards contain less information than dense rewards. But, it should always be possible to design sparse reward strategies, regardless of the task.

2.2.7. Accelerating RL

Because sparse rewards contain less information than dense rewards, it is necessary to accelerate the RL learning process. One common method for accelerating RL is to use *reward shaping*, which involves the creation of complex reward functions, where the RL agent is rewarded for accomplishing tasks that are associated with accomplishing the ultimate goal. One example of reward shaping is a reward strategy for the game of chess where the RL agent is rewarded for capturing pieces. Unfortunately, reward shaping strategies are often exploited by RL agents, which prevents the RL agent from learning to reach the ultimate goal [37]. In the chess example, an RL agent may never learn to checkmate an opponent if it always prioritizes capturing pieces. For this reason, we chose not to use reward shaping. Instead, we used *curriculum learning* and *transfer learning*.

"Curriculum learning" refers to the process of teaching an RL agent to reach the ultimate goal by first learning to reach easier goals. This mimics the process of human learning, which is greatly facilitated by a curriculum (e.g., teach arithmetic before calculus). In the context of circuit design, we implemented curriculum learning by asking the RL agent to build shorter circuits, and then we progressively increased the length of the circuits.

"Transfer learning" refers to the process of teaching an RL agent to perform one task and then using that RL agent to perform another task. In the context of circuit design, we implemented transfer learning by teaching the RL agent to build shorter circuits, and then we used the RL agent to build very long circuits. We differentiate between curriculum learning and transfer learning by noting that curriculum learning is a progressive process and transfer learning is a sudden process.

For instance, in curriculum learning, we might teach an RL agent to build a 20-component circuit by slowing increasing the circuit length from 10-20 in increments of 1. In transfer learning, we might teach the RL agent to build a 20-component circuit by starting with circuits that contain 10 components, and then by abruptly increasing the number of circuit components to 20.

2.2.8. Software Implementation

The RL framework was implemented using in Python 3.6.8 using stable-baselines3 version 1.3.0 [40]

3. EXPERIMENTS FOR RL-BASED CIRCUIT DESIGN

In this chapter, we list the outcomes of the experiments that guided the development of the RL-enhanced system design tool. A summary of the experiments is provided in Table 3-1. Full summaries of these experiments can be found in Appendix B.

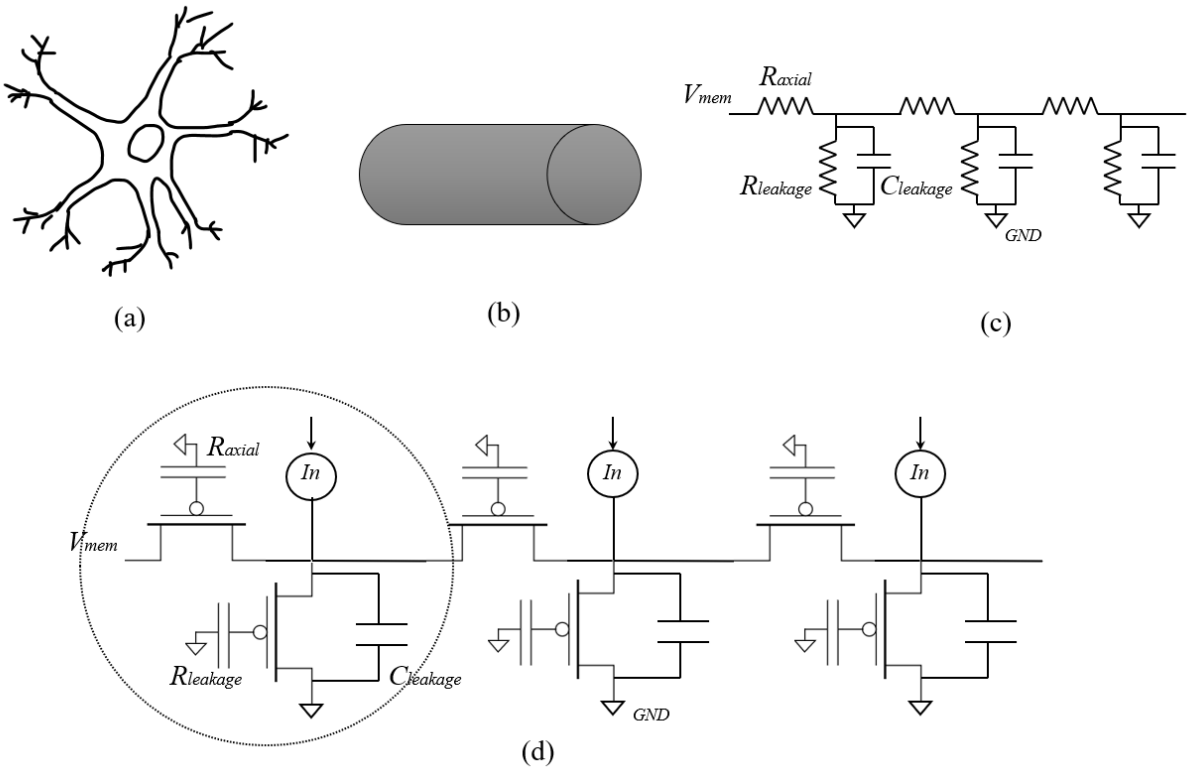


Figure 3-1. (a) Dendrites are the structures which connect synapses to the cell body and are known to perform interesting computation like summation of inputs, non-linear filtering, and coincidence detection for spatio-temporal inputs. (b) Neuroscientists typically model these structures as passive linear cables. (c) The classical model of this linear cable is an equivalent RC delay line. (d) A VLSI implementation [35, 13] of this model using MOSFETs (Metal-Oxide-semiconductor field-effect transistor) and capacitors in the sub-threshold regime. The circled structure is similar to our abstracted model for the sub-circuit.

To evaluate our framework, we focused on the dendrite neural circuit. Dendrites are highly branched tree-like structures that connect a neuron's synapses to the soma. Research shows that dendrites can perform operations such as non-linear filtering, spatial and temporal summation of synaptic inputs, coincidence detection, synaptic scaling, and sequence detection [26]. Biological

#	Type	Topic	Purpose
1	delay	simple delay line	simplest interesting example of RL-enhanced design
2	delay	delay line with 2 delay types	simplest interesting example of multi-component RL-enhanced design
3	delay	delay line with continuous parameters	choose discreet and continuous actions simultaneously
4	delay	delay line with costs	optimize performance while minimizing costs
5	delay	delay line with time series input	the RL agent observes entire time series instead of single time steps
6	detect	simple delay gate - dense rewards	generalize delay line to detection line
7	detect	simple delay gate - sparse rewards	use sparse rewards instead of dense rewards
8	detect	simple delay gate - curriculum learning	use curriculum learning to accelerate learning
9	detect	delay gate with variable delays	detection task that does not require detection at each time step
10	detect	delay gate with analog detectors	detection task with simultaneous choice of discrete and real-valued parameters
11	dendritic	dendritic delay line (no leaks)	simplest generalization from detection to dendritic detection
12	dendritic	dendritic delay line (leaky)	confirm that RL works with leaky dendritic compartments
13	dendritic	dendritic delay line with tunable bias	improve detection by making bias tunable
14	dendritic	dendritic delay line with tunable leak	attempt to improve detection by making the leakage tunable

Table 3-1. Summary of RL-enhanced design experiments.

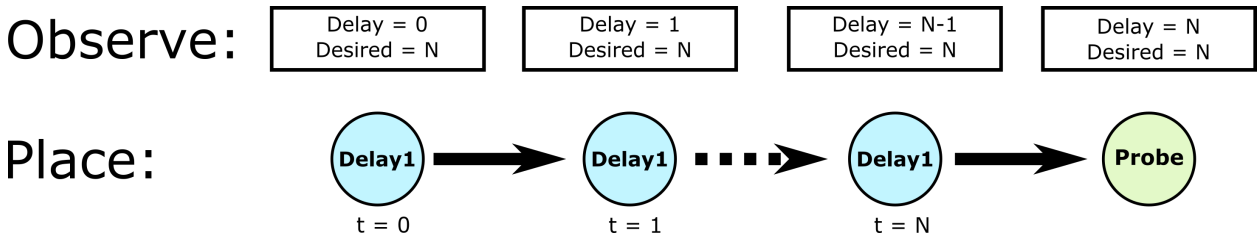


Figure 3-2. Illustration of experimental setup for Experiments 1 and 5. For Experiment 1, the delays were presented as integers. For Experiment 5, the delays were presented as delta functions.

dendrites are also known for their complex physical structures with significantly greater fan-in ($\sim 10,000$ inputs).

Analog approaches to building dendrites offer savings in energy compared to digital neuromorphic, with the promise of more complexity and dynamics. Previous work has demonstrated dendritic models using sub-threshold analog floating-gate transistors [13, 35] as shown in Fig. 3-1. The voltage across these devices has been shown to obey a class of non-linear partial differential equations (PDEs) and could potentially be used as scientific kernels. Our hypothesis is that the technology to implement nonlinear summation currently exists, but capturing this capability will require specific configuration of the neural network. Automating the selection of parameters for this circuit will enable circuit designers to predict how to utilize these circuits in architecture design.

3.1. Delay Line Experiments

3.1.1. Experiment 1: Simple Delay Line

We trained an RL agent to perform the delay line task (Section 2.1.1) with variable delays that lasted between 0 and 5 time steps. At each time step, the RL agent chose to either place a Delay1 component or a Probe component, which terminated the circuit (Table 2-1). The RL agent was given a sparse reward (Equation 2.2). The RL agent was trained using HER-DQN (Section 2.2). After training, the RL agent was able to create delay line circuits with 100% accuracy. This experiment demonstrated that RL can be used to design simple circuits from scratch. For an illustration, see Figure 3-2.

3.1.2. Experiment 2: Delay Line with Two Delay Types

We trained an RL agent to perform the delay line task (Section 2.1.1) with variable delays that lasted between 0 and 100 time steps. The RL agent was only allowed to place a maximum of 11 components. At each time step, the RL agent chose to either place a Delay1 component, a Delay10 component, or a Probe component, which terminated the circuit (Table 2-1). The RL agent was given a sparse reward (Equation 2.4). The RL agent was trained using HER-SAC

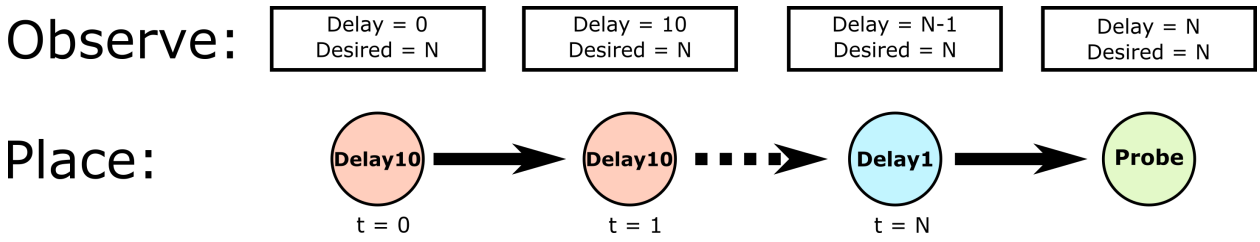


Figure 3-3. Illustration of experimental setup for Experiment 2.

(Section 2.2). After training, the RL agent was able to create delay line circuits with 100% accuracy. For an illustration, see Figure 3-3.

This experiment demonstrated several important concepts. Firstly, it taught us to log-transform rewards so that small and large delays were equally considered. Secondly, it taught us to penalize too much delay so that RL agents would learn to not make unfixable mistakes. Thirdly, it demonstrated that RL methods for real-valued action spaces can also be used to make discreet component choices. And finally, it demonstrated that RL could make simple circuits from more than 2 component types.

3.1.3. Experiment 3: Delay Line with Continuous Parameters

We trained an RL agent to perform the delay line task (Section 2.1.1) with variable delays that lasted between 0 and 100 time steps. The RL agent was only allowed to place a maximum of 11 components. At each time step, the RL agent chose to either place a Delay1 component, a Delta Delay component, or a Probe component, which terminated the circuit (Table 2-1). The RL agent was given a sparse reward (Equation 2.4). The RL agent was trained using HER-SAC (Section 2.2). After training, the RL agent was able to create delay line circuits with 100% accuracy. This experiment demonstrated that RL agents can learn to not only place components, but also choose their parameter values. For an illustration, see Figure 3-4.

3.1.4. Experiment 4: Delay Line with Costs

We trained an RL agent to perform the delay line task (Section 2.1.1) with variable delays that lasted between 0 and 100 time steps. The RL agent was only allowed to place a maximum of 11 components. At each time step, the RL agent chose to either place a Delay1 component, a Delta

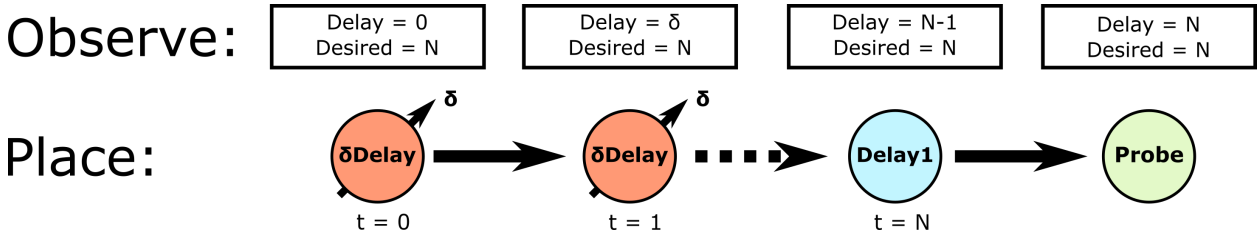


Figure 3-4. Illustration of experimental setup for Experiments 3 and 4.

Delay component, or a Probe component, which terminated the circuit (Table 2-1). The RL agent was given a sparse reward (Equation 2.5) that included costs for each component placed. The RL agent was trained using HER-SAC (Section 2.2). After training, the RL agent was able to create delay line circuits with 100% accuracy. This experiment demonstrated that RL agents can learn to maximize circuit performance while simultaneously minimizing some cost. For an illustration, see Figure 3-4.

3.1.5. Experiment 5: Delay Line with Time Series Input

We trained an RL agent to perform the delay line task (Section 2.1.1) with variable delays that lasted between 0 and 10 time steps. The RL agent was only allowed to place a maximum of 11 components. At each time step, the RL agent chose to either place a Delay1 component or a Probe component, which terminated the circuit (Table 2-1). The RL agent was given a sparse reward (Equation 2.4). The RL agent was trained using HER-SAC (Section 2.2).

For this experiment, the observation was different. Whereas in previous experiments, the RL observed the remaining delay at a discreet point in time, for this experiment, the RL agent observed a time series that represented the necessary delay at each time step. The RL agent also observed an integer representing the number of steps left. Unfortunately, training was not successful. It is possible that longer training times would have resulted in successful training. Future experiments should also consider how alternative neural network architectures can be used to understand simultaneous presentation of time series data. It is possible that recurrent neural networks will be useful for such problems. For an illustration, see Figure 3-2.

3.2. Detection Experiments

3.2.1. Experiment 6: Simple Delay Gate - Dense Rewards

We trained an RL agent to perform the detection task (Section 2.1.2) with variable signal lengths between 0 and 10 time steps. At each time step, the RL agent observed a sample of the Digital signal (Figure 2-4). At each time step, the RL agent chose to either place a Detect0 component, a Detect1 component, or a Probe component, which terminated the circuit (Table 2-2). The RL agent was given a dense reward (Equation 2.8). The RL agent was trained using PPO (Section

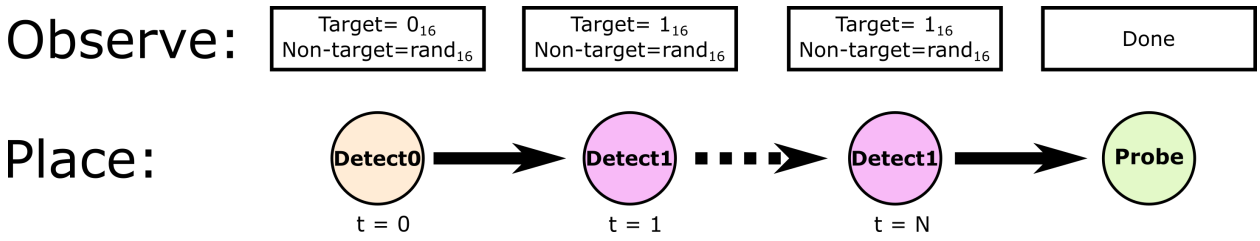


Figure 3-5. Illustration of experimental setup for Experiments 6, 7, and 8.

2.2). After training, the RL agent was able to build circuits that correctly detected the target pattern with nearly 100% accuracy. For an illustration, see Figure 3-5.

This experiment confirmed that RL can learn to design signal detection circuits by looking at multiple examples of target an non-target signals. Until this experiment, RL agents observed scalar values, rather than vectors.

3.2.2. Experiment 7: Simple Delay Gate - Sparse Rewards

We trained an RL agent to perform the detection task (Section 2.1.2) with variable signal lengths between 0 and 10 time steps. At each time step, the RL agent observed a sample of the Digital signal (Figure 2-4). At each time step, the RL agent chose to either place a Detect0 component, a Detect1 component, or a Probe component, which terminated the circuit (Table 2-2). Unlike in the previous experiment, the RL agent was given a sparse reward (Equation 2.8). The RL agent was trained using PPO (Section 2.2). For an illustration, see Figure 3-5.

When the target signal had fewer than 5 time steps, the RL agent was able to learn to build an appropriate signal detection circuit. However, when the target signal had more than 5 timesteps, the RL agent did not learn to build signal detection circuits within the allotted time. This experiment highlighted the need to accelerate RL when using sparse rewards.

3.2.3. Experiment 8: Simple Delay Gate - Curriculum Learning

We trained an RL agent to perform the detection task (Section 2.1.2) with variable signal lengths between 0 and 10 time steps. At each time step, the RL agent observed a sample of the Digital signal (Figure 2-4). At each time step, the RL agent chose to either place a Detect0 component, a Detect1 component, or a Probe component, which terminated the circuit (Table 2-2). The RL agent was given a sparse reward (Equation 2.8). The RL agent was trained using PPO (Section 2.2). For an illustration, see Figure 3-5.

For this experiment, curriculum learning was applied by increasing the maximum length of the target signal as training progressed. The length of the target signal increased from 0 to 10. At the end of training, the RL agent was able to design signal detection circuits that were nearly 100% accurate for all signal lengths. This experiment highlighted the utility of using curriculum learning to accelerate RL when using sparse rewards. Because curriculum learning was so successful, it was used for the remainder of experiments.

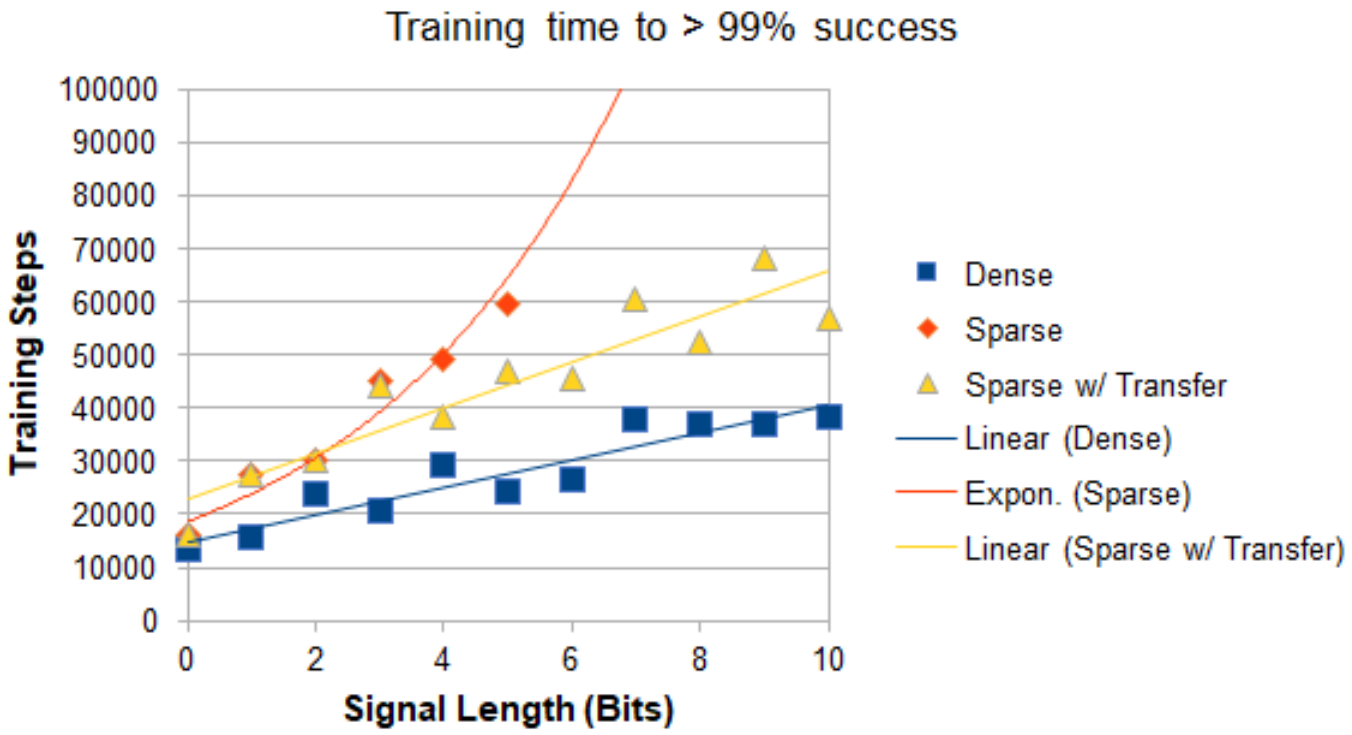


Figure 3-6. Curriculum learning. RL agents were trained for 100,000 timesteps to design signal detection circuits. When RL agents were given dense rewards (Experiment 6, blue line), training time increased linearly with the signal length. When RL agents were given sparse rewards (Experiment 7, red line), training time increased exponentially with the signal length, and RL agents did not learn to design signal detection circuits for signals that contained more than 5 bits of information. When RL agents were trained using curriculum learning, training time increased linearly with the signal length, even with sparse rewards (Experiment 8, yellow line).

3.2.4. Experiment 9: Delay Gate with Variable Delays

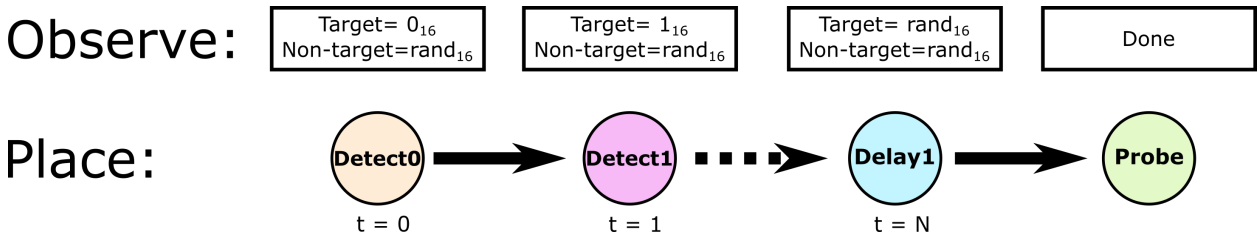


Figure 3-7. Illustration of experimental setup for Experiment 9.

We trained an RL agent to perform the detection task (Section 2.1.2) with variable signal lengths between 0 and 10 time steps. At each time step, the RL agent observed a sample of the Digital signal (Figure 2-4). At each time step, the RL agent chose to either place a Detect0 component, a Detect1 component, a Delay1 component, or a Probe component, which terminated the circuit

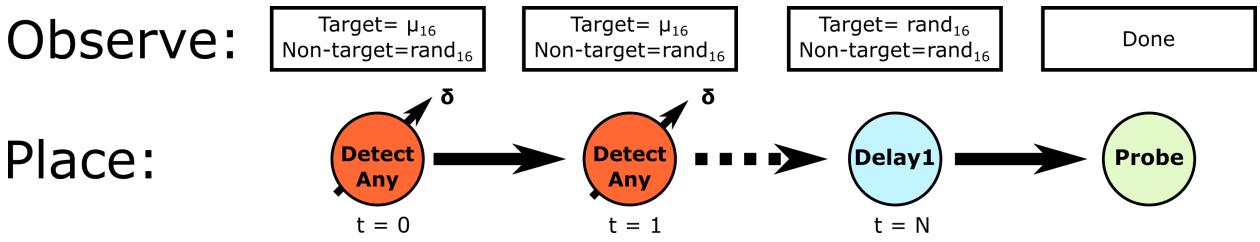


Figure 3-8. Illustration of experimental setup for Experiment 10.

(Table 2-2). The agent was given a sparse reward (Equation 2.8). The RL agent was trained using PPO (Section 2.2). For an illustration, see Figure 3-7.

This experiment was much like Experiment 8, but with the addition of the Delay1 component, which did not perform detection. The addition of the Delay1 component was made to test if the RL agent was able to learn to place components when there were more than 3 components. This experiment established that increasing the design space does not necessarily inhibit the performance of the RL agent.

3.2.5. Experiment 10: Delay Gate with Analog Detectors

We trained an RL agent to perform the detection task (Section 2.1.2) with variable signal lengths between 0 and 10 time steps. At each time step, the RL agent observed a sample of the Low SNR signal (Figure 2-4). At each time step, the RL agent chose to either place a Detect Any component or a Probe component, which terminated the circuit (Table 2-2). The RL agent was given a sparse reward (Equation 2.8) and trained using PPO (Section 2.2). For an illustration, see Figure 3-8.

After training, the RL agent was able to build signal detection circuits that were greater than 60% accurate. Transfer learning was employed by training the agent to build signal detection circuits for signals of length 10 and then using the same agent to build signal detection circuits for signals of length 20, without retraining. Transfer learning worked with little to no degradation in performance (about 2%). While accuracy was far from perfect, it was far greater than chance performance (50%), and somewhat impressive given the low SNR.

This experiment established the possibility of using transfer learning to train an RL agent in a time-efficient manner on simple circuits and then using the same agent to build more complex circuits. This experiment also established that RL agents can be used to build real-valued signal detection circuits. Rather than choosing to tune performance for the Detect Any component, we choose to shift attention to the dendritic detect components. It is possible that tuning of the curriculum learning hyperparameters or additional training time would have resulted in higher performance.

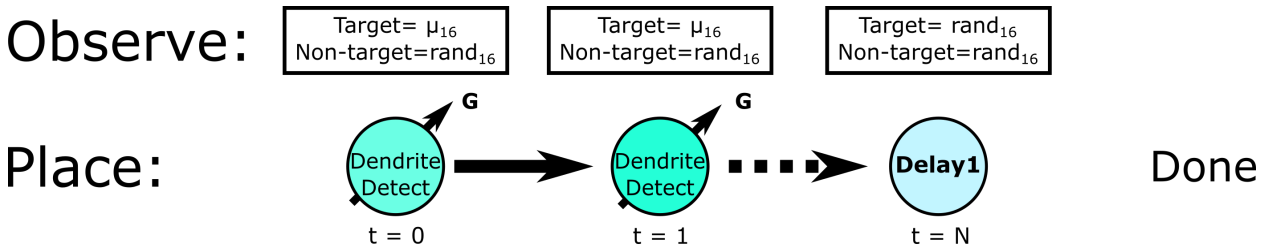


Figure 3-9. Illustration of experimental setup for Experiment 11.

3.3. Dendritic Detection Experiments

Passive dendrite cable circuits are a great first step to test our RL algorithms on. Each dendrite compartment or 'tap' consists of an axial conductance and leakage capacitance and conductance. These blocks can be placed sequentially and can be tuned for a particular parameter.

3.3.1. Experiment 11: Dendritic Delay Line (No Leaks)

We trained an RL agent to perform the dendritic detection task (Section 2.1.3) with variable signal lengths between 10 and 20 time steps. At each time step, the RL agent observed a sample of the Medium SNR signal (Figure 2-4). At each time step, the RL agent chose to either place a Dendritic Detect component or a Delay component. Unlike in previous experiments, circuits terminated automatically based on the pre-determined number of components in the circuit; probe components were not used. The RL agent was given a sparse reward (Equation 2.8) and trained using PPO (Section 2.2). For an illustration, see Figure 3-9.

Detection accuracy approached 91% for the Medium SNR signal. For this task, we noticed that the highest accuracy was achieved when target and non-target signals had samples with opposite signs, suggesting that a tunable bias parameter may be helpful. We also noticed that this task required slightly longer signals (a minimum length of about 10 was used) in order for the RL agent to be able to learn. This may be because only certain differences can be exploited by dendritic components.

Curriculum learning was successfully used to increase the signal length from 10 to 20. Transfer learning was successfully used to increase the signal length from 20 to 40. This experiment demonstrated that RL can be used to build circuits from dendrite-like neuromorphic components.

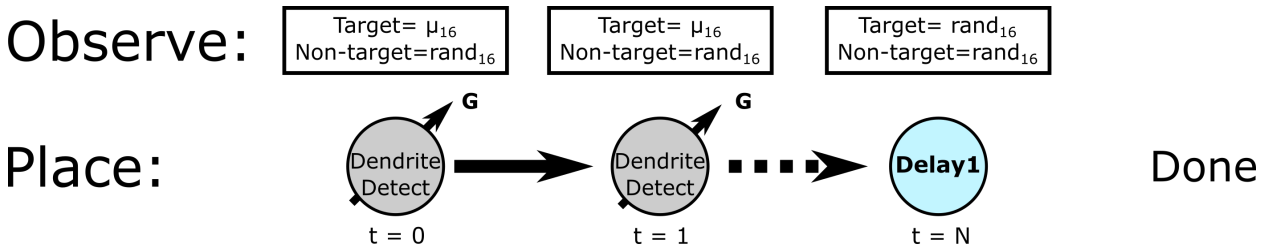


Figure 3-10. Illustration of experimental setup for Experiment 12.

3.3.2. Experiment 12: Dendritic delay line (leaky)

We trained an RL agent to perform the dendritic detection task (Section 2.1.3) with variable signal lengths between 10 and 20 time steps. At each time step, the RL agent observed a sample of the Medium SNR signal (Figure 2-4). At each time step, the RL agent chose to either place a Leaky Dendritic Detect component or a Delay component. Circuits terminated automatically based on the pre-determined number of components in the circuit; probe components were not used. The RL agent was given a sparse reward (Equation 2.8) and trained using PPO (Section 2.2). For an illustration, see Figure 3-10.

Detection accuracy approached 91%. Curriculum learning was successfully used to increase the signal length from 10 to 20. Transfer learning was successfully used to increase the signal length from 20 to 40. This experiment demonstrated that RL can be used to build circuits from dendrite-like neuromorphic components, even if those components have leakage currents.

3.3.3. Experiment 13: Dendritic Delay Line with Tunable Bias

We trained an RL agent to perform the dendritic detection task (Section 2.1.3) with variable signal lengths between 10 and 20 time steps. At each time step, the RL agent observed a sample of the Medium SNR signal (Figure 2-4). At each time step, the RL agent chose to either place a Leaky Dendritic Detect with Tunable Bias component or a Delay component. Circuits terminated automatically based on the pre-determined number of components in the circuit; probe components were not used. The RL agent was given a sparse reward (Equation 2.8) and trained using PPO (Section 2.2). For an illustration, see Figure 3-11.

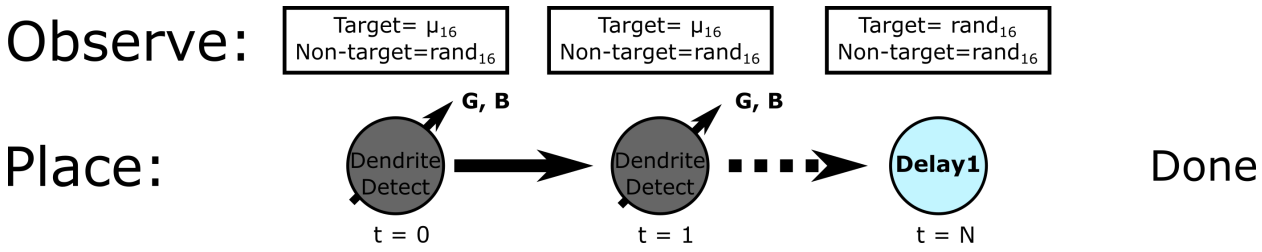


Figure 3-11. Illustration of experimental setup for Experiment 13.

Detection accuracy approached 97%. Curriculum learning was successfully used to increase the signal length from 10 to 20. Transfer learning was successfully used to increase the signal length from 20 to 40. This experiment demonstrated that RL can be used to rapidly prototype changes in components. In this case, including the tunable bias increased accuracy from 91% to 97%.

3.3.4. Experiment 14: Dendritic Delay Line with Tunable Leak

We trained an RL agent to perform the dendritic detection task (Section 2.1.3) with variable signal lengths between 10 and 20 time steps. At each time step, the RL agent observed a sample of the Medium SNR signal (Figure 2-4). At each time step, the RL agent chose to either place a Leaky Dendritic Detect with Tunable Leakage component or a Delay component. Circuits terminated automatically based on the pre-determined number of components in the circuit; probe components were not used. The RL agent was given a sparse reward (Equation 2.8) and trained using PPO (Section 2.2). For an illustration, see Figure 3-12.

Detection accuracy approached 91%. Curriculum learning was successfully used to increase the signal length from 10 to 20. Transfer learning was successfully used to increase the signal length from 20 to 40. Leakage current is a part of biological neurons. Here, we rapidly prototyped a tunable leakage current to determine if it had any affect on computation. Compared to a non-tunable leakage current (Experiment 12), a tunable leakage current did not offer additional computational power.

3.4. Discussion

In this chapter, we prototyped RL circuit design tools on simple delay line tasks before moving to signal detection tasks and finally dendritic detection tasks. From these experiments, we learned several important lessons. Firstly, RL can successfully design many different types of circuits from scratch by choosing components from a library. The maximum size of the library that we tested was 4 components, but we suspect that these methods will generalize to larger libraries. Secondly, RL can maximize circuit performance while minimizing costs. Thirdly, RL can use sparse rewards to learn to design circuits; however, when using sparse rewards, it is important to accelerate RL by using techniques such as curriculum learning and transfer learning. Lastly, RL is very useful for rapid prototyping. In less than an hour, it was possible to evaluate several dendrite-like neuromorphic components to determine which components would be useful for performing a signal detection task (see Experiments 11-14).

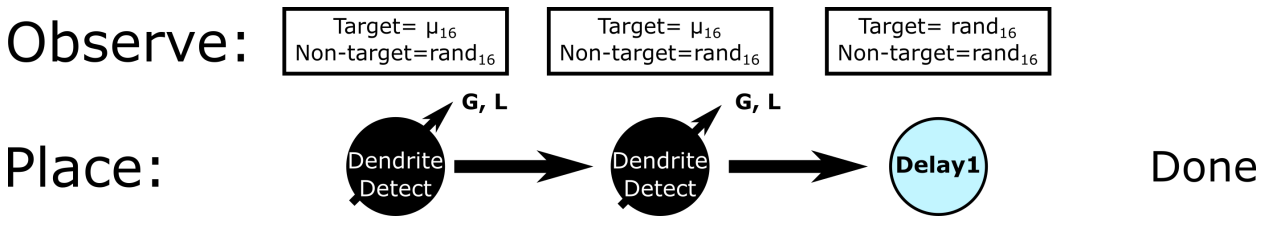


Figure 3-12. Illustration of experimental setup for Experiment 14.

These experiments also uncovered one unaddressed challenge that future studies should consider: the RL agents were unable to build circuits when they observed an entire time series input instead of observing a time series sample-by-sample. Many interesting electrical components, such as capacitors, induce phase changes, and detecting these phase changes may be crucial for allowing RL to place such components. For this reason, it will be necessary for RL to be able to observe multiple samples of a time series. Future studies should consider methods that enable time series observations - recurrent neural networks are one potential solution.

4. COMPARING REINFORCEMENT LEARNING WITH EVOLUTIONARY ALGORITHMS

4.1. Motivation

In Chapter 3, RL was successfully used to build signal detection circuits. In this chapter, we explore if evolutionary algorithms (Section 1.6.3) can be used to accelerate the circuit design process.

4.2. Methods

4.2.1. Task

We compared evolutionary algorithms and RL using the Dendritic Detection task. The Dendritic Detection task was described at length in Section 2.1.3. Briefly, models of dendrite-like neuromorphic components were use to construct signal detection circuits. Circuits were only constructed from the Dendritic Delay and Dendritic Detect with Tunable Bias components (Table 2-3). These circuits were built to detect the Medium SNR signals (Figure 2-4). RL agents were given sparse rewards (Equation 2.4).

4.2.2. Details of Evolutionary Algorithm

Unlike RL, evolutionary algorithms do not have a mechanism to observe system inputs, outputs, or design constraints. Instead, evolutionary algorithms optimize circuits by using the fitness score (reward), only.

For the Dendritic Detection task, the evolutionary algorithm was allowed to optimize the following parameters:

1. The identity of each component
2. The parameters of each component

The number of components was assumed to be fixed (this was also the case for the RL algorithm). Component identity and component parameters were combined such that crossover (Section 1.6.3) chose the component identity and component parameters from the same parent (i.e., it was not possible to inherit a component identity from one parent and inherit the component parameters from the other parent). Crossover was implemented by randomly choosing the component parameter/identity traits with equal probability from each of the 2 parents. Mutation

was implemented by randomly changing the component identities and parameters. Component identities were mutated with a probability of 0.1. Component parameters were mutated with a probability of 0.1 and were drawn from a uniform distribution that was centered on the original parameter value with a range of ± 1. Note that the range of parameter values was ± 10.

Each generation was composed of 50 parents. Crossover and mutation produced an additional 50 children. The top 50 candidates were selected to become the next generation.

4.3. Results

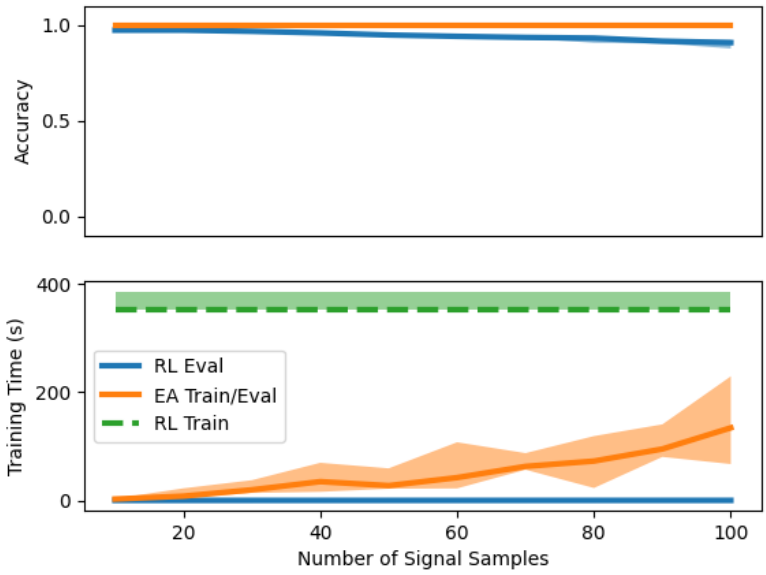


Figure 4-1. Reinforcement learning vs. evolutionary algorithms. Evolutionary algorithms (orange line) outperformed RL (blue line) in terms of detection accuracy (top panel) for all signal lengths. Detection accuracy for the evolutionary algorithm was always perfect, while the detection accuracy for RL decreased with the number of samples in the signal. The training time (bottom panel) for the evolutionary algorithm (orange line) increased with the number of samples in the signal. For RL, the training time was constant for all signal lengths (green line), and circuit design time (blue line) increased as a function of signal length, but remained much lower than the training time for evolutionary algorithms.

Figure 4-1 compares the performance and training/evaluation time required by RL and EA.

4.3.1. Reinforcement Learning

For RL, it took approximately 6 minutes (median) to train the agents (n=5). Because of transfer learning, training time was constant for all circuit sizes tested. Note that, for the results detailed in

Figure 4-1, the RL agents were trained on circuits that were 10-20 samples in length and then tested on circuits that were 10-100 samples in length. After training, inference (circuit design without further learning) occurred at a rate of approximately 1,269 components per second. This rate was relatively constant for all circuit sizes. The median classification accuracy of circuits designed with RL was approximately 97% for circuits that had 10-20 modules and greater than 90% for circuits that had up to 100 modules.

4.3.2. Evolutionary Algorithms

When EA is used to optimize circuits, there is no distinction between training time and testing time because EA must be performed anew for each new circuit design. Unlike RL, which learns to build a class of circuits (in this case, dendrite-like neuromorphic circuits), EA learns to build individual circuits. Thus, circuit design time for EA is dependent on circuit size. As circuit size was increased from 10 to 100 modules, the design time increased from a median of 2.5 seconds/circuit to a median of 134 seconds/circuit. The classification accuracy of circuits designed with EA was 100%, regardless of circuit size.

4.4. Discussion and Conclusions

4.4.1. Circuit Design Time

RL took approximately 6 minutes to train, but then it could design dendrite-like neuromorphic signals at a rate of 1,269 modules/second. EA had no distinct training phase, and could build circuits at a rate of 0.5-4 modules/seconds. This suggests that EA and RL may have distinct roles in AI-enhanced circuit design. EA can be used for rapid prototyping with novel modules, and RL can be used for "production" circuit design that involves designing many different circuits from the same modules. However, we suggest caution when interpreting these results for several reasons. Firstly, hyperparameters for EA and RL were manually tuned, and we expect that additional hyperparameter tuning may result in accelerated learning for both methods.

Secondly, while EA has already been used to design networks of modules with parallel connections [45], generalizing RL to parallel circuits is non-trivial and will require further research. This means that EA may have a distinct advantage when designing more complex circuits. However, RL is able to take advantage of transfer learning to efficiently learn more complex circuits. Designing transfer learning methods for EA is likely not possible without specialized knowledge. This means that RL may have advantages over EA when circuit design tasks can be broken down into series of less complex tasks.

4.4.2. Circuit Design Accuracy

As shown in Figure 4-1, EA was able to build circuits that could detect signals with 100% accuracy, whereas RL-designed circuits were only able to achieve a median accuracy of 90-97%, depending on signal length. This might suggest that EA should be used instead of RL when high

accuracy is desired. Or, alternatively, it might suggest that RL solutions can be refined using EA in order to combine the superior speed of RL (during inference) with the superior accuracy of EA.

However, it is important to note that the target and non-target distributions that were used to create signals overlapped, suggesting that, at least occasionally, signal detection should have imperfect accuracy. Perfect accuracy over the 3200 signals evaluated suggests that the 100% accuracy for EA-designed circuits is caused by overfitting. We note that, while RL simulations automatically provide validation data sets, EA methods are difficult to cross-validate. For this reason, it may be preferable to use RL instead of EA because RL provides a reasonable estimate of solution robustness whereas EA does not.

4.4.3. Circuit Design Creativity

EA must be trained separately for each circuit of interest. RL can be trained separately for each circuit, but such a strategy would be uncommon and likely quite inefficient. Instead, RL is typically trained on a large set of circuits that belong to the same family (e.g., the family of signal detection circuits that are built from specified modules for signals with the specified distributions). Because RL is trained to build many different circuits, it is encouraged to find solutions that work well across many types of circuits. In many regards, this behavior is desirable, because it suggests that, if some RL solutions can be explained, then most RL solutions can be explained. However, because EA is trained on a single circuit, it can possibly overfit the data (as discussed above). EA also can produce multiple designs that work well for an individual circuit. A trained RL agent, by contrast, will tend to produce solutions deterministically. Because EA can produce multiple designs, it can be thought of as a creative process. If more creativity is desired, the EA can be randomly initialized multiple times in order to reveal the entirety of the solution space. By understanding the entire solution space, it may be possible to promote discovery of novel uses for devices.

Creative processes tend to be inefficient, and EA is no exception. An RL agent is expected to produce the same circuit design each time that it is presented with the same example signals (assuming that no additional training occurred). While different RL agents could produce different circuit designs, this becomes less likely as the amount of training data increases because there are relatively few solutions that work well across the entire problem space, and low-performing RL agents would be discarded. EA, on the other hand, can produce multiple circuit designs across runs, and even within the same run. While this is great for creativity, it comes at the cost of efficiency. It is difficult to know if further training for an EA algorithm will produce better results. Thus, when EA does not produce a solution that has 100% accuracy on the training data, it is possible to waste time searching for a better solution. A trained RL agent, on the other hand, produces a single solution; so, it is not necessary to spend additional time to search for a better solution.

5. CASE STUDY: MOTT MEMRISTORS

5.1. Motivation

In previous chapters, we showed that RL and evolutionary algorithms can be used to rapidly optimize circuit designs. In this chapter, we demonstrated that AI-enhanced circuit optimization tools can be used to rapidly prototype circuits using emerging devices. Mott memristors [23] were recently developed to implement leaky integrate-and-fire dynamics, similar to biological neurons. Here, we show that evolutionary algorithms can rapidly tune Mott memristor parameters to perform a signal detection task.

5.2. General Methods

5.2.1. Mott Memristor Model

Memristors are resistors, where the resistance is a function of the history of current that has passed through the device. In a Mott memristor, the change in resistance is based on a temperature-based Mott transition, which causes the device to transition from a resistor to a conductor.

Here, we modeled Mott memristors as variable resistors. The variable resistance was modulated by the history of voltage across the device. In our model, the value of the resistor could take on one of two values. If a voltage was applied to the Mott memristor that exceeded the switching voltage, v_s, the memristor transitioned from a high resistance value, r_h, to a low resistance value, r_l. In our model, v_s and r_l were tunable, but r_h was not tunable. Mott memristors can be operated with either forward or reverse polarity by physically rotating the memristor relative to the flow of current. While a positive v_s is required to switch the memristor from r_h to r_l, a negative v_s can be implemented by switching the polarity of the device.

Mott memristors like those described in [23] also have a capacitance that acts a charge integrator. Here, we modeled this capacitance as a perfect integrator of voltage that was capable of instantaneously storing and dissipating charge (and therefore, voltage). Integration of voltage was calculated as:

$$v_T = \sum_{t=0}^{T} v_t \tag{5.1}$$

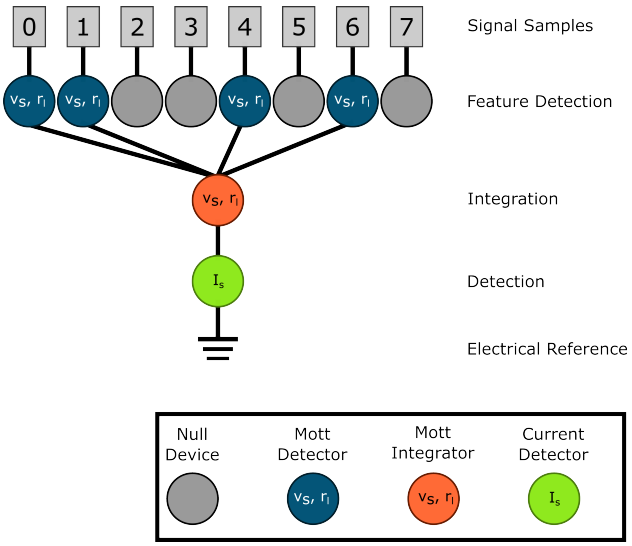


Figure 5-1. Mott signal discrimination circuit. Mott detectors (blue) are volatile Mott memristors that are used to detect samples where the target signal differs from the non-target signal. Null devices (gray) are essentially empty component spaces that are not involved in detection. Mott integrators (red) are non-volatile Mott memristors that integrate information across time from multiple Mott detectors. Current detectors (green) make a decision about whether or not a signal has been detected.

Note that, in our model, leakage current was not present. However, we did assume that, for the non-volatile Mott memristors (the Mott Integrator modules, as described below), the internal state of the device was reset between detection events.

We used two different models of Mott memristors. In both models, a voltage greater than v_s caused the resistance of the device to transition from r_h to r_l. In "volatile" memristor models, the resistance of the device transitioned from r_l to r_h when voltage decreased below v_s. In "non-volatile" memristor models, the resistance of the device transition from r_l to r_h only when the polarity of the voltage changed to the opposite polarity of the memristor.

To test if AI techniques could be used to build circuits from Mott memristors, we asked the AI to build circuits that could perform signal detection tasks. Because evolutionary algorithms were identified as good candidates for rapid prototyping in Chapter 4, we chose to use EA to optimize the circuit design. We used the same evolutionary algorithm methods that were described in Section 4.2 with a few modifications. Firstly, we decreased the number of samples in the signal from 10 to 8. This is because the Mott memristor circuits require 1 Mott memristor per signal sample plus 2 additional elements that are involved in integrating information across samples (see Figure 5-1). We also chose to use the High SNR signal distributions (Figure 2-4) in order to rapidly prototype this initial proof-of-concept.

5.2.2. Circuits

Evolutionary algorithms were used to optimize Mott memristor signal detection circuits like the one illustrated in Figure 5-1. The evolutionary algorithm had to choose the component identity and parameters from the following:

1. **Mott Detector** - a volatile Mott memristor with tunable parameters v_s and r_l
2. **Null Device** - a module that accepts a sample of the input signal, but does not process it.
3. **Mott Integrator** - a non-volatile Mott memristor with tunable parameters v_s and r_l
4. **Current Detector** - an ideal current detector (no resistance), with tunable current detection threshold parameter I_s

The connectivity of all elements was assumed to be constant and known. Mott Detectors were always connected to a single Mott integrator, which was always connected to a Current Detector. The evolutionary algorithm had to tune all tunable parameters to cause the Current Detector to detect target signals, but not non-target signals.

5.2.3. Tunable Parameter Values

The following Mott memristor parameters were tunable:

$$v_s \in [0, 10] \tag{5.2}$$
$$r_l \in [-10, 10] \tag{5.3}$$

The sign of the r_l determined the polarity of the Mott memristor. The current detection threshold was also tunable for most experiments:

$$I_s \in [0, 10] \tag{5.4}$$

For some experiments, $I_s = 0.5$ was a fixed constant in order to investigate if evolutionary algorithms could take advantage of the Mott transition for computation.

5.3. Results

5.3.1. Tunable Current Detection Threshold

The evolutionary algorithm was asked to perform the signal detection task, described above, and allowed to choose I_s from the range [0, 10]. Under these conditions, EA converged to solutions quickly (within 50 generations) and never failed to find a solution that resulted in 100% accuracy.

Interestingly, the I_s was always approximately 0 and never observed to be greater than 0.1. Furthermore, the solutions found by EA never required the Mott memristors to transition from r_h to r_l. Under these simulation conditions, all identified solutions used Mott memristors as simple resistors.

One EA-produced solution for the signal detection task with tunable current detectors is shown in Figure 5-2. The evolutionary algorithm placed Mott detectors at samples 4, 5, and 7. With this solution, the current detector was unable to perform signal detection until all 3 samples had been observed (e.g., signal detection could not detect signals until sample 7 was observed).

5.3.2. Fixed Current Detection Threshold

The evolutionary algorithm was asked to perform the signal detection task, described above, and forced to use $I_s = 0.5$. Under these conditions, the evolutionary algorithm was able to find solutions in approximately 1 out of 3 trials. Whenever solutions were found, they resulted in 100% classification accuracy.

Unlike in the tunable current detector condition, the fixed current detector condition did find solutions that required the Mott memristors to transition from r_h to r_l.

One example of the signal detection task with fixed current detectors is shown in Figure 5-2. The solution found by the evolutionary algorithm used Mott detectors at samples 0-3 and 6-7. However, the current detector was able to detect the signal using only signal samples 0 and 1. Once the signal was detected, the trial terminated. This demonstrates that, when permitted by differences in the signal and noise distributions, evolutionary algorithms can build circuits that do not require using the full signal to perform detection. However, the final design did include more Mott Detectors than required. We note that no feedback was given to the evolutionary algorithms to discourage it from using more Mott Detectors than necessary. Future work should consider adding a penalty to the fitness score to encourage finding solutions that use fewer Mott Detectors. Or, future solutions might consider a second refinement step that removes as many Mott Detectors as possible (replacing them with Null Modules) until performance is affected.

5.4. Discussion

In this chapter, we demonstrated that evolutionary algorithms can be used to optimize circuits that use emerging devices. Future studies should use more complete Mott memristor models. Future studies should also consider if RL can increase the proportion of fixed current detector circuits ($I_s = 0.5$ condition) that can be successfully optimized.

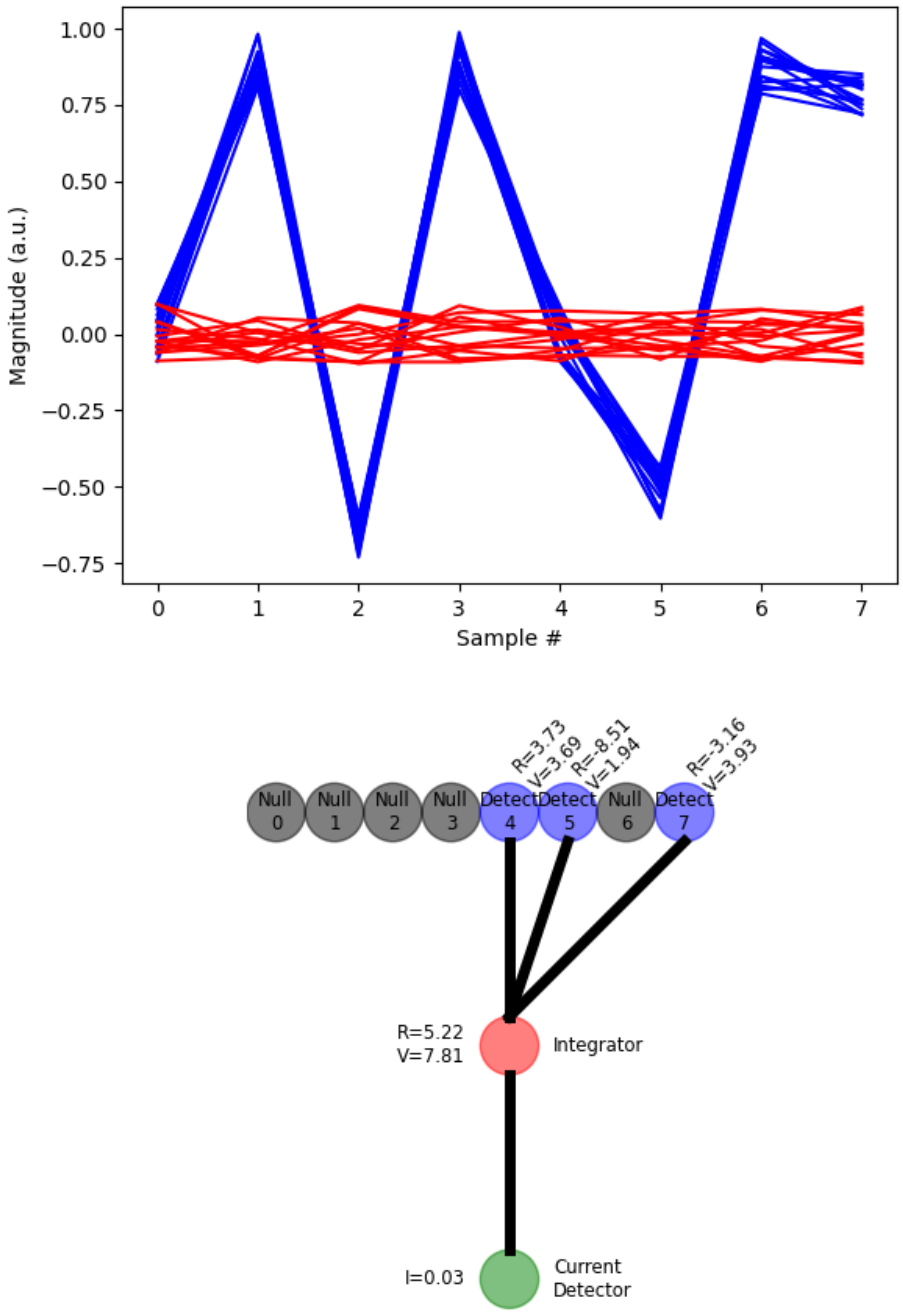


Figure 5-2. Signal detection with tunable current detectors. (Top Panel) An example of a signal detection task . Blue lines are the target signal and red lines are the non-target signals. (Bottom Panel) The solution found by the evolutionary algorithm.

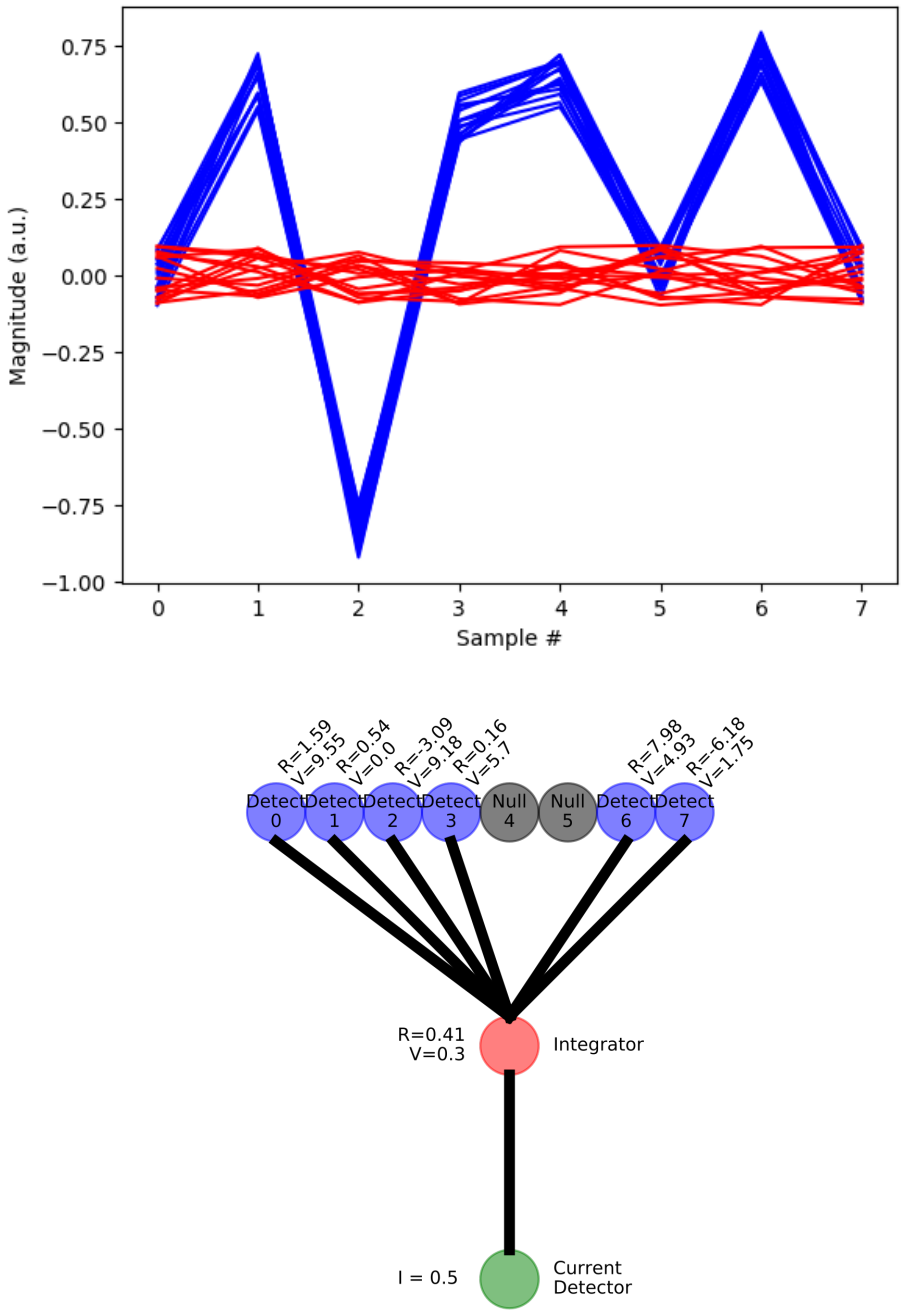


Figure 5-3. Signal discrimination with fixed current detectors. (Top Panel) An example of a signal detection task . Blue lines are the target signal and red lines are the non-target signals. (Bottom Panel) The solution found by the evolutionary algorithm.

6. NOVEL DEVICE DISCOVERY

6.1. Motivation

In Chapters 3-5, we demonstrated that AI-enhanced circuit design tools can be used to quickly design circuits from components that are contained in a library. This rapid prototyping capability allows investigators to quickly identify problems that an emerging device can solve. However, the process of design relies on a library of known components. But suppose that no combination of known components solves a problem within the specified design constraints? In this situation, investigators need to move beyond design and into the discovery of novel devices. There are at least 2 situations where novel device discovery should be implemented:

1. The user is an engineer trying to solve a circuit design problem, but the performance of the final circuit does not meet minimum performance requirements.
2. The user is a scientist who is interested in discovery.

Unfortunately, the process of discovery can take a long time - possibly years depending on the investigator's level of experience and inspiration. Here, we present an AI-enhanced circuit design tool that is capable of producing specifications for novel devices. In theory, specifications for novel devices can be used to actually fabricate the device. In order to demonstrate what one such design process would look like, we present this chapter as a chronological process of discovery.

6.2. General Methods

In this section, we describe a general design process that can be followed for novel device development.

6.2.1. Defining the Problem

The process of discovery relies on MAMMAL (Section 2.2.5). MAMMAL requires the investigator to supply circuit inputs, desired circuit outputs, and any design constraints. If MAMMAL is provided with an insufficient component library, it can switch to "discovery mode." For this example, we chose to provide MAMMAL with an empty component library.

Circuit components are functions that accept a finite number of inputs and produce a finite number of outputs. These functions may also accept several constant parameters. In order to discover a novel component, MAMMAL needs to describe a useful function. Once a function is described, investigators can attempt to implement the function using novel or existing devices or materials.

6.2.2. Neural Network Models

To discover useful functions, we need a technique that can approximate useful functions. For this purpose, we used neural networks, which are universal function approximators.

As show in Figure 6-1, MAMMAL chooses modules from a library. If MAMMAL chooses the "novel component" component, the component is represented by a neural network. Importantly, all novel component neural networks have the same neural network parameters. These shared parameters include w_1 (layer 1 weights), w_2 (layer 2 weights), μ (layer normalization mean), and σ (layer normalization standard deviation), as shown in Figure 6-1. We refer to these parameters as "neural network parameters." Shared neural network parameters ensure that device behavior is consistent across all instances of the novel component. This behavior is desirable for at least 2 reasons:

1. Intuitively, all instances of a novel component should exhibit the same behavior, at a high level (e.g., all capacitors store charge).
2. Shared parameters should decrease the time required to construct new components because there are fewer parameters to optimize.

However, MAMMAL was allowed to choose non-shared input parameters that it could pass to each neural network separately. These non-shared parameters are represented in Figure 6-1 as θ. The number of input parameters can be chosen by the investigator. We refer to the non-shared parameters as "componentence" parameters (componentence is to components as resistance is to resistors as capacitance is to capacitors). In theory, a componentence parameter is a device parameter (such as the distance between capacitor plates) or a material property (such as a dielectric constant). While shared neural network parameters ensure that novel component behavior is consistence across components at a high level, non-shared componentence parameters allow for customization of device behavior at a low level (e.g., capacitors all store charge, but each capacitor stores a different amount of charge).

6.2.3. Training

The process of device discovery involves iterative training of MAMMAL (which selects componentence parameters) and training of the neural network novel component models. Once the neural network component models are trained, they can be analyzed to extract the desirable behavior of a novel component.

6.2.4. Analysis Methods

We propose two methods to analyze the neural network models to understand component behavior.

The first method involves analyzing the distribution of the componentence parameters chosen by the RL agent. Examples of this analysis are shown in Figures 6-3, 6-5, 6-7, and 6-9. From this analysis, we can derive information about:

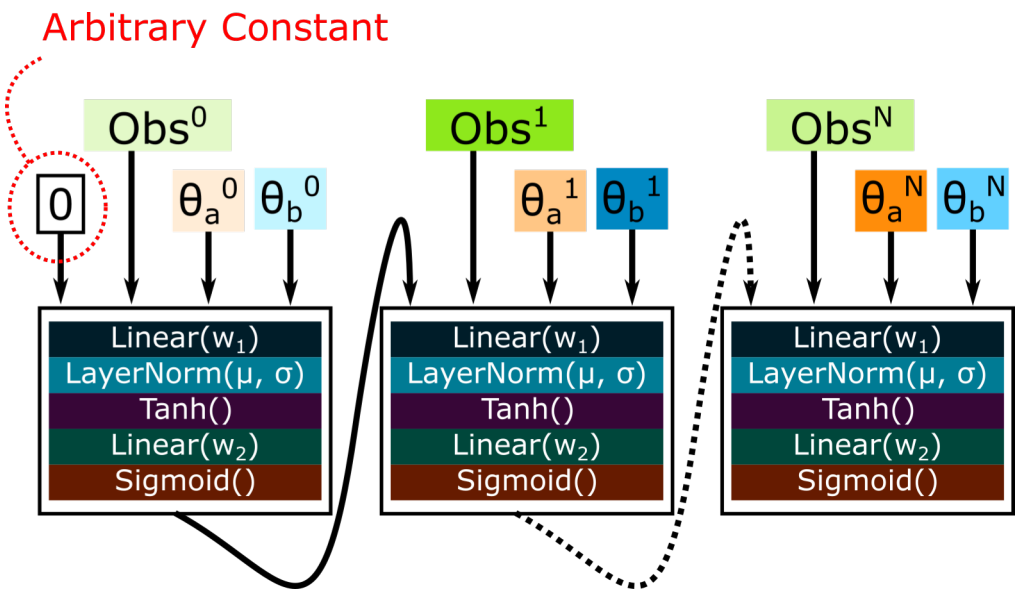


Figure 6-1. Example of a neural network for device discovery. Neural networks are composed of multiple modules, where each module represents a single instance of a component. All neural network parameters (w_1, w_2, μ, σ) are shared between neural networks. Because the neural network parameters are constant between components, each component will have the same behavior, given the same inputs. Each neural network receives the output of the previous component, as well as possible additional inputs (Obs.) and non-shared componentence parameters (θ_a, θ_b) that are chosen by MAMMAL. The θ parameters are different between instances of the components, which allows the same neural network to perform slightly different computations, depending on context. Input 1 for the first component is an arbitrary constant.

1. The number of componentence parameters that are necessary to perform the specified task.
2. The range of componentence parameters that will need to be manufactured.
3. The sensitivity of the component to changes in componentence.

The second method involves analyzing the input-output characteristics of the component. For a component with a single input, a single output, and a single componentence parameter, the input-output characteristics can be displayed on a single plot where the x-axis is the input, the y-axis is the componentence, and the z-axis (color axis) is the output. For components that have 2 inputs, 2 componentence parameters, and a single output, results can be displayed as a grid of subplotplots, where each subplot represents a combination of the 2 componentence parameters, the x- and y-axes of each subplot represent the 2 inputs, and the output of the component is displayed on the z-axis (color axis). Examples of such plots are shown in Figures 6-4, 6-6 , 6-8, 6-10. From such plots, we can extract useful information about the function implemented by the component and how the function of the component is changed by the componentence parameters.

6.2.5. Task

The RL agent was trained to perform the Separable About Zero signal discrimination task (Figure 2-4) using sparse rewards (Equation 2.8). The RL agent was trained using PPO (Section 2.2).

6.2.6. Human-Intuitive Solution

To assist in interpreting the input-output behavior of novel devices, we present an example of novel device behavior that a human might try to invent. We note that, for the Separable About Zero task, the target and non-target signals always have opposite signs. Therefore, a human would likely try to solve this problem by creating a component that has one of two componentence values: {-1, 1}. A componentence parameter of -1 would correspond with a negative number detector, and a componentence parameter of 1 would correspond with a positive number detector. The input-output behavior of such a device is demonstrated in Figure 6-2. Note that this solution is very similar to the solution implemented in RL Experiment 8.

6.3. “Novel” Device Generation - Initial Attempt

We attempted to design a novel device by allowing the RL agent to specify 2 componentence parameters. Final performance exceeded 99% accuracy.

In Figure 6-3, the distribution of the componentence parameters is plotted. In the left subplot, the axis limits were chosen to maximize the resolution of the interesting data. In the right subplot, the axis limits were chosen to show the distribution of the componentence parameters over the entire allowed search space. There is clear structure in the distributions of the componentence parameters.

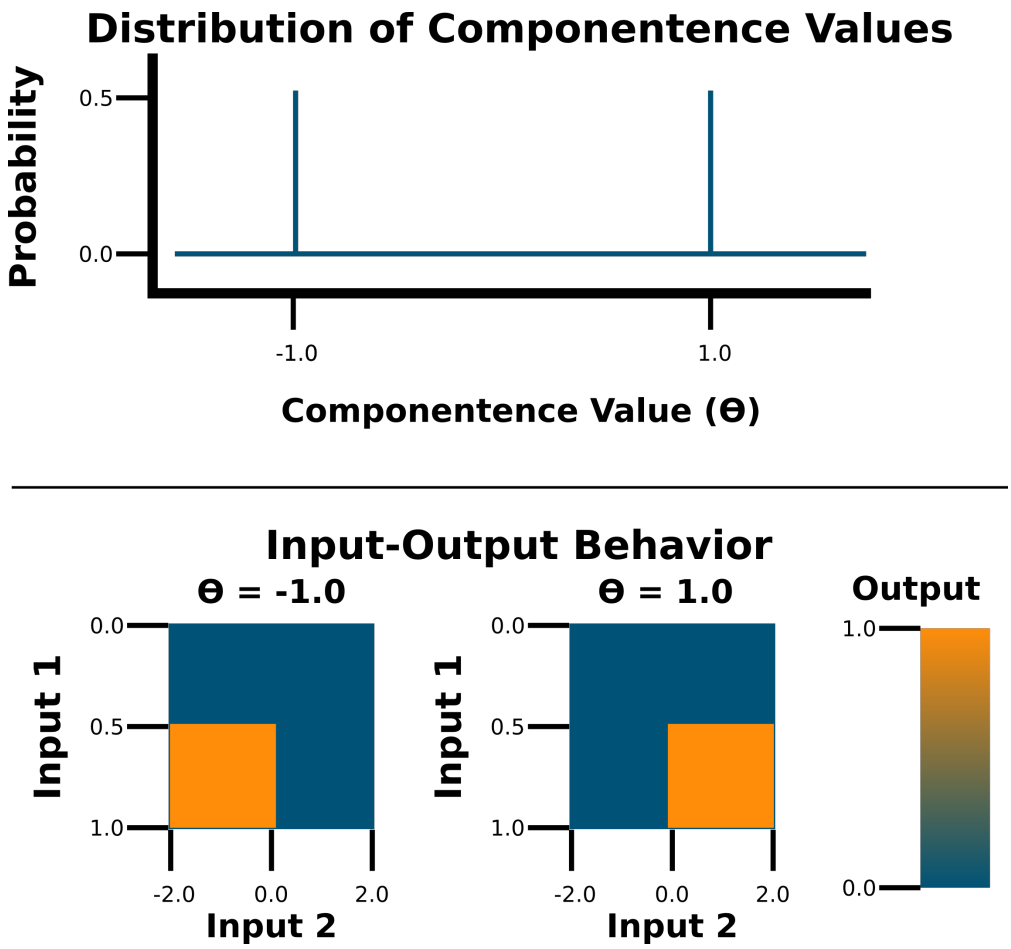


Figure 6-2. Example of human-intuitive component behavior for the Separable About Zero task. In the top panel, we see that the component has 2 intuitive componentence values — one to detect positive signs, and one to detect negative signs. In the bottom panel, we see the input output behavior. With a componentence value of $\theta = -1$, the component is a negative number detector, which only outputs a one if the previous component output a 1 and if Input 2 is negative (see Figure 2-2 for an illustration of a detection line). With a componentence value of $\theta = 1$, the component is a positive number detector, which only outputs a one if the previous component output a 1 and if Input 2 is positive.

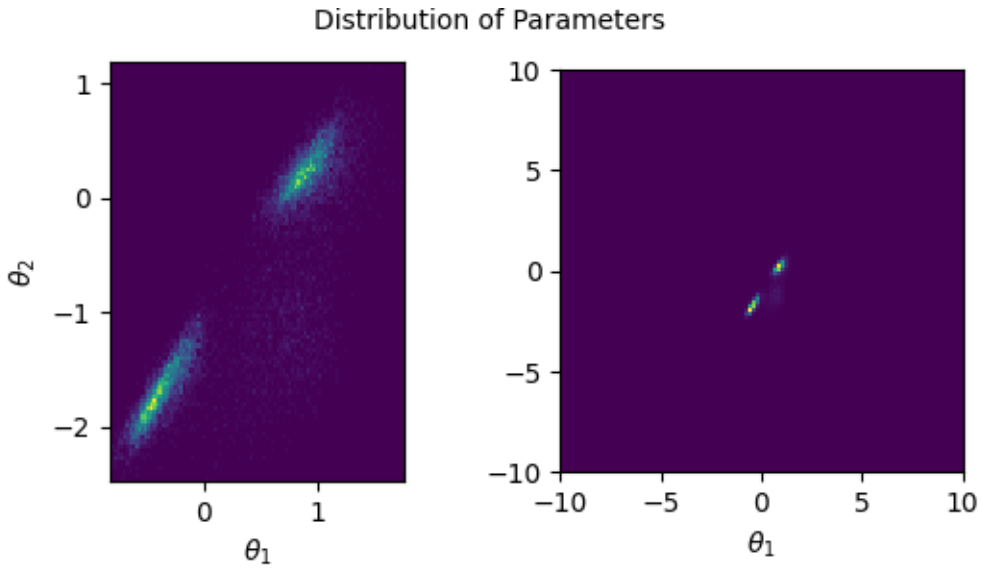


Figure 6-3. Distribution of the θ_1 and θ_2 componentence parameters for a neural network model of a novel neuromorphic component. The right panel is a plot of the componentence parameters across their entire range. The left panel is a detailed view of the right panel.

From the right subplot, it is clear that the input parameters that were chosen by MAMMAL only made up a small portion of the permitted range of the componentence parameters. Componentence parameters were within the approximate range of [-2, 2], even though they were permitted to be chosen from the range [-10, 10]. Because the range of the parameters was limited, future experiments could potentially accelerate learning by restricting the componentence space that needs to be explored.

From the left subplot, it is clear that the parameters θ_1 and θ_2 are both bi-modal. The bi-modal distributions of the θ parameters strongly suggests that the componentence parameters do not need to be continuous-valued - the problem can likely be solved with parameters that are chosen from a set of only 2 possible parameter values.

Additional inspection reveals that θ_1 and θ_2 co-vary; the joint distribution lies on the line described by the $\theta_2 = \theta_1$. These observations, taken together, suggest that we do not need 2 componentence parameters to solve the given problem.

As shown in Figure 6-4, the output of the novel component (colors) can be plotted as a function of the inputs (x- and y-axis) as well as the componentence parameters (each subplot). By analyzing the subplots, we see that the component acted like a classifier. The decision line becomes more linear, and less cubic, as the θ_1 componentence parameter increases. The θ_2 componentence parameter also seems to make the decision line more linear, but it also seems to shift the decision line towards zero (note the inverted axis).

Input 1 values were restricted to the range [0, 1] by the sigmoidal output of the neural network. Input 2 parameters are plotted over the range [-2, 2], but the target and non-target signals that are observed on Input 2 were limited to the range [-1, 1]. Thus, when analyzing the behavior of the

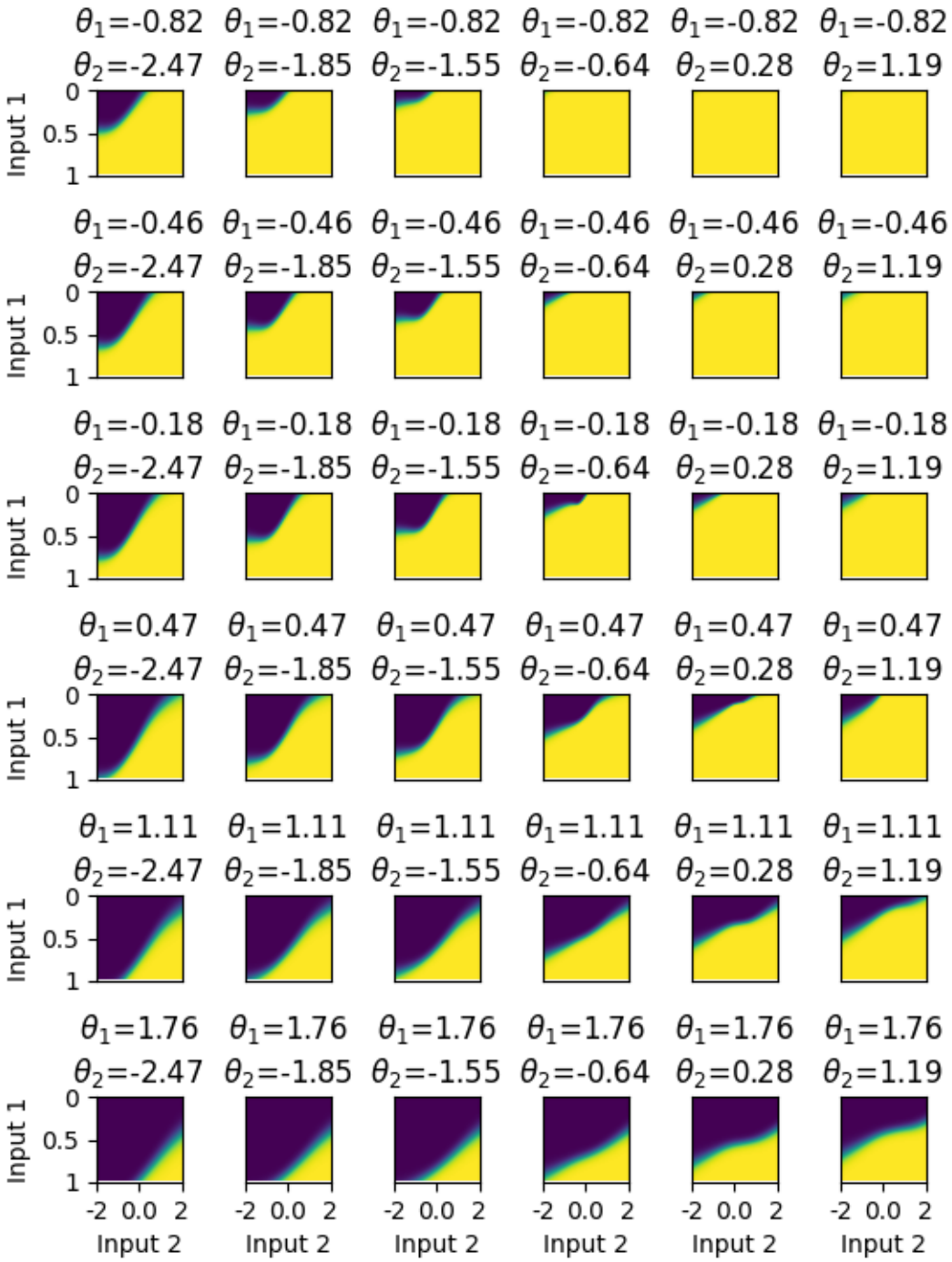


Figure 6-4. Analysis of component model outputs as functions of the component inputs and the componentence parameters.

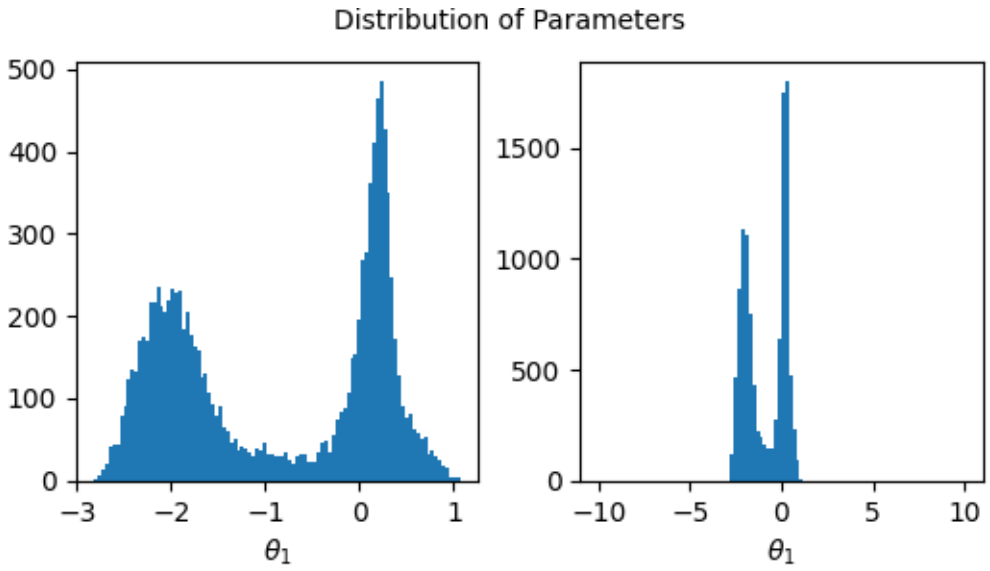


Figure 6-5. Distribution of θ_1 for a neural network model of a novel neuromorphic component. Only one componentence parameter was used.

component, it is important to note that Input 2 values in the range [-1, 1] are interpolations, while Input 2 values greater than 1 in magnitude are extrapolations. This provides an interesting way to analyze the robustness of the neural network model.

6.4. Decreasing the Number of Componentence Parameters

During the first attempt to generate novel devices, we determined that fewer than 2 componentence parameters were required to solve the problem (Figure 6-3). In theory, the problem requires at least 1 componentence parameter to solve (Section 6.2.5). Thus, for the second attempt to generate a novel device, we allowed the RL agent to choose only a single parameter. Final performance was not affected by this simplification - performance continued to exceed 99% accuracy.

In Figure 6-5, we see that the distribution of the componentence parameters chosen by the AI-enhanced circuit design tool remained bi-modal - indicating that the componentence parameter likely needs to only take on one of two values. This is consistent with our analysis in Section 6.2.6. We also see that the componentence parameters take up a small portion of the allowable domain (i.e., the allowable domain was [-10, 10], but the componentence values chosen were in [-3, 1] (x-axis)).

In Figure 6-6 we can analyze the affects of the componentence parameter θ_1 on the input-output behavior of the novel component. The dividing line appears as a cubic function, which seems to shift upwards and become more linear as the value of θ_1 increases from its minimum observed value to its maximum observed value.

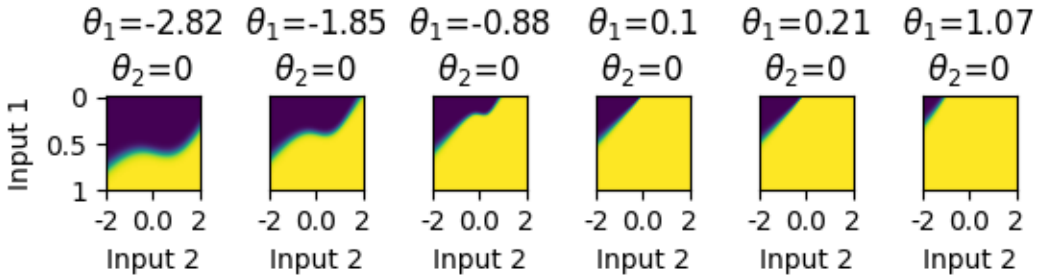


Figure 6-6. Analysis of component model outputs as functions of the component inputs and the componentence parameters. Unlike in Figure 6-4, we only fit one neural network import parameter.

We note that the distribution of the componentence parameters is bi-modal with modes at $\theta_1 \approx -2, 0.21$. We can divide our analysis into analysis of each of the distribution modes.

For the left-most mode ($\theta_1 \approx -2$):

The component outputs a 1 whenever it detects very large positive numbers on Input 2, or when it detects small-magnitude positive or negative values on Input 2.

For the right-most mode ($\theta_1 \approx 0.21$):

The component always outputs a 1 if Input 1 (from the previous component) is > 0.5 or if Input 2 (observation) is positive. Negative numbers can only be detected if Input 1 is greater than 0.

This is an interesting strategy that differs from the human-intuitive strategy (Section 6.2.6). The human-intuitive strategy distinctly detects positive and negative numbers. The novel component strategy seems to have one component that detects positive numbers and one component that "doesn't stop" negative number values from preventing the detection of a signal.

We notice that the left and right sub-distributions (centered around each of the two modes) have approximately equal area, indicating that the "2" componentence values are chosen with approximately equal probability. This makes sense, in some regards, since 50% of the target signal samples are positive and 50% of the target signal samples are negative (and vice-versa for the non-target samples).

6.5. "Correcting" the Problem Formulation

In the previous analysis of Figure 6-5, we concluded that the area under the curve for the left sub-histogram was approximately equal to the area under the curve for the right sub-histogram. This indicated that each of the "2" componentence values was chosen with approximately equal probability. However, further analysis also reveals that the left sub-histogram is shorter and fatter than the right sub-histogram, suggesting that, to function properly, the "left" componentence value needs to vary more. Analysis of Figure 6-6 reveals that the decision boundary has an

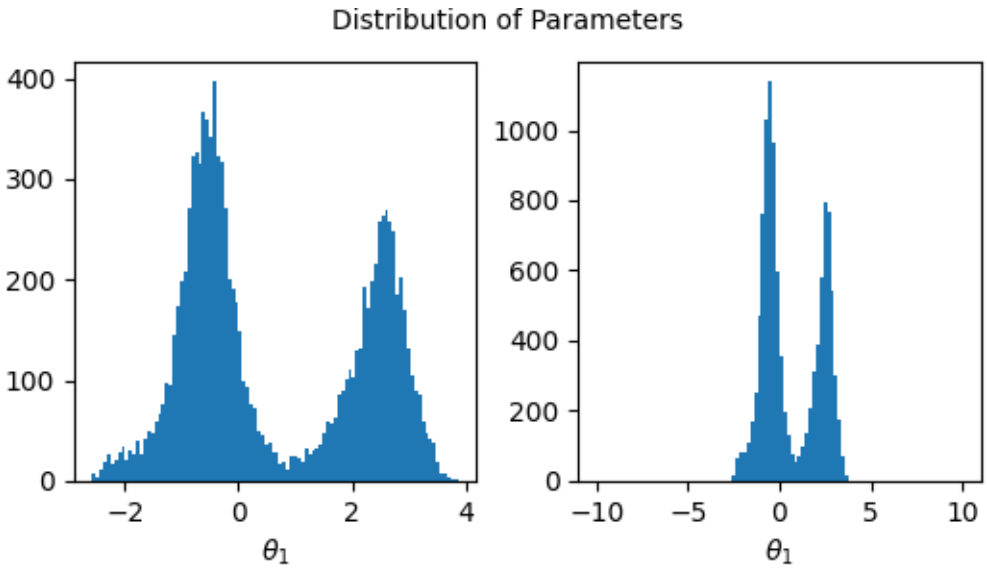


Figure 6-7. Distribution of θ_1 for a neural network model of a novel neuromorphic component. Input 1 for the initial component was corrected to 1.

approximately cubic shape when θ_1 is drawn from the sub-distribution centered on the left-most node. A cubic shape seems overly complicated for such a simple discrimination task.

Further analysis reveals that the componentence distribution in Figure 6-5 is not centered at 0. Certainly, the distribution of componentence values was not constrained to be centered at 0. However, this seems like an oddly complicated result for such a simple discrimination task, and it certainly clashes with the human-intuitive solution (Section 6.2.6), which was very much centered at 0.

Of course, the neural network model was not constrained to produce human-intuitive results. However, these observations led us to re-analyze the problem description. During our re-analysis, we discovered an odd choice. As shown in Figure 6-1, Input 1 is a number in the range [0, 1] that approximately represents whether the previous components detected the target signal (1) or the non-target signal (0). The range [0, 1] is enforced by the sigmoidal activation function at the output of the previous component. The first neural network component doesn't have a "previous component," meaning that Input 1 needs to be a specified (but otherwise arbitrary) constant, rather than a variable. As shown in Figure 6-1, we had chosen to set the specified constant to 0 for the initial neural network component. This would cause the human-intuitive solution to fail (Section 6.2.6) because the human-intuitive solution assumes that all previous components detected a target signal and, therefore, output 1's. Despite this choice, the novel components were able to be used to classify most signals correctly.

It is possible that setting the initial value of Input 1 to 0 influenced the componentence parameter distributions and the shape of the decision line. To investigate this idea, we "discovered" another component, but this time, we set the initial value of Input 1 to 1. Final performance was not greatly affected by this decision - performance continued to exceed 98% accuracy.

As shown in Figure 6-7, the sub-histograms corresponding to each of the two values of θ_1 are

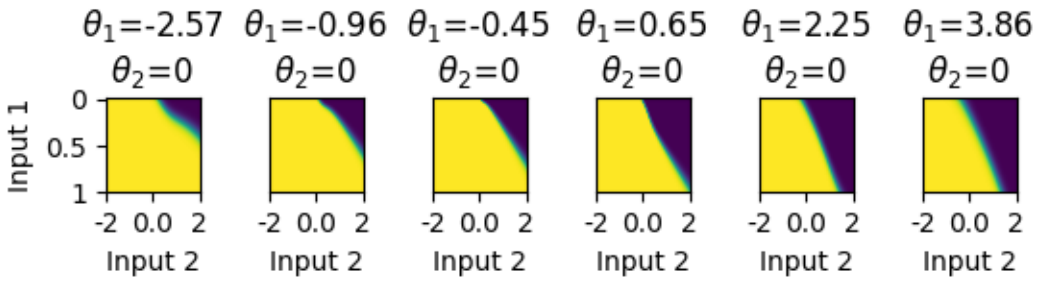


Figure 6-8. Analysis of component model outputs as functions of the component inputs and the component parameters. Input 1 for the initial component was corrected to 1.

more equal, though not perfectly so. Additionally, the distribution is much closer to being centered at zero, though, again, not perfectly so. As shown in Figure 6-8, the line dividing the output-1-region and the output-0 region is now linear, regardless of the value taken by θ_1. All of these differences seem to agree better with the human-intuitive solution.

The input-output behavior was different than the behavior of the component that was "discovered" in the previous section. Not only was the decision boundary much simpler, but the slope of the decision boundary was reversed. When θ_1 is negative, the component always outputs a 1. When θ_1 is positive, the component output a 1 only if it detected a "lack of a positive number." This strategy can be thought of as "positively identifying a target signal, unless it looks a lot like a non-target signal." This is in contrast to the component characterized in Figures 6-5 and 6-6 which "negatively identified a target signal, unless it looked a lot like a target signal."

From this attempt to design a novel component, we learned the following:

1. By analyzing the componentence parameters, as well as the input-output characteristics of novel components, we can gain insight into how components actually need to function. This provides some hope that black box neural network models of novel components can actually be transformed into novel hardware.

2. "Small" changes in problem statements can lead to big differences in component behavior. This property is wonderful for device discovery.

In closing, we would like to re-emphasize the fact that the "mistake" discovered in this section (in addition to the previous section) was discovered as part of an actual case study. By characterizing the input-output characteristics of a black box model of a novel component, we were able to gain insight into its actual function.

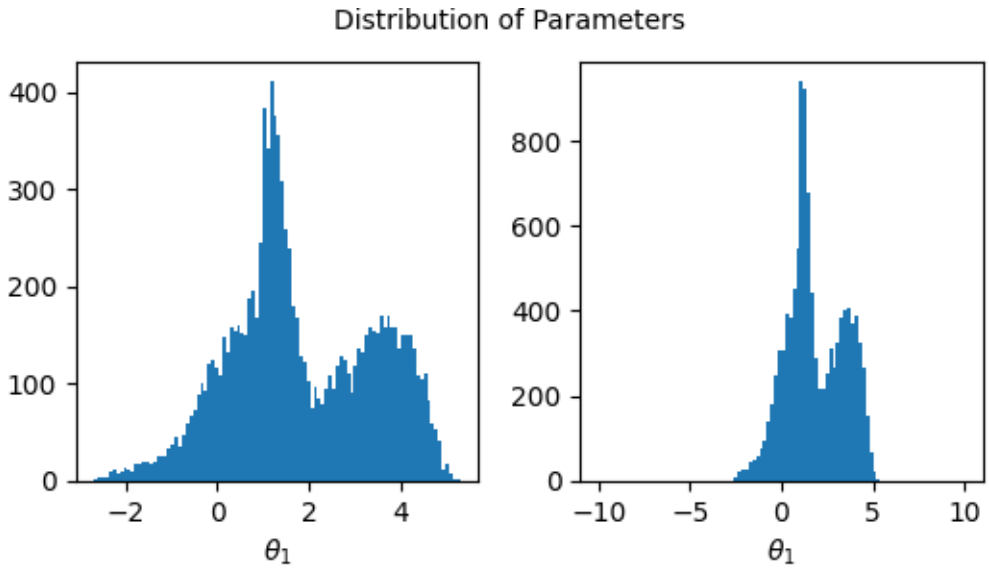


Figure 6-9. Distribution of θ_1 for a neural network model of a novel neuromorphic component. Input 1 for the initial component was changed to 0.5.

6.6. Expanding the Library

In the previous attempt to create a novel component, we changed the value of Input 1 for the initial neural network component from 0 to 1. We observed that this change greatly affected the input-output characteristics of the resulting component, while minimally affecting the performance. In an effort to discover more interesting components, we set the value of Input 1 for the initial neural network component to 0.5. Final performance was not greatly affected by this change - performance continued to exceed 98% accuracy.

By analyzing the distribution of the componentence parameter (Figure 6-9), we see that the distribution is no longer centered at 0, which supports the previous hypothesis that a non-zero componentence parameter mean is associated with a departure from the human-intuitive problem solution. We observe that the componentence parameter distribution remains bi-modal, but the left-most mode is more than twice as likely as the right-most mode. Furthermore, we note that the inter-modal region of the distribution has a relatively high magnitude compared to previous distributions of componentence parameters.

When analyzing the input-output behavior of the component, we see that the behavior of the component is quite complex compared to the behavior of previous components. At a high level, θ_1 seems to shift the decision boundary downward and scale the decision region around its "point." For low values of θ_1, the component "detects" everything as a signal - essentially acting like a simple delay component (Table 2-2). As the value of θ_1 increases, the component begins to behave like the positive number detector described in Section 6.2.6 and shown in Figure 6-2.

For values of θ_1 surrounding the left-most mode, the component seems to detect low-magnitude positive and negative values. For values of θ_1 surrounding the right-most mode, the component seems to detect positive values, but, it can detect values around 1 better than it can detect values

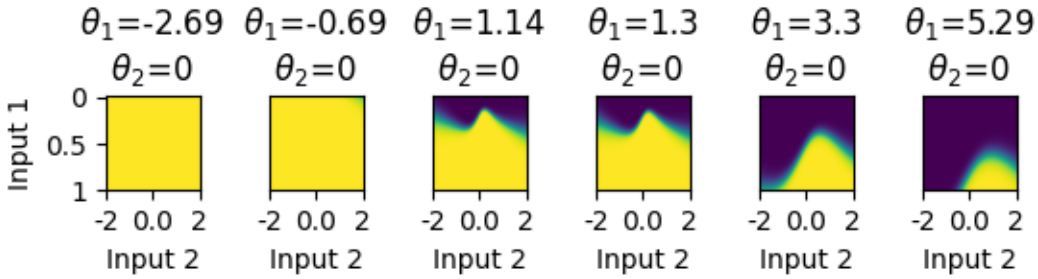


Figure 6-10. Analysis of component model outputs as functions of the component inputs and the component parameters. Input 1 for the initial component was changed to 0.5.

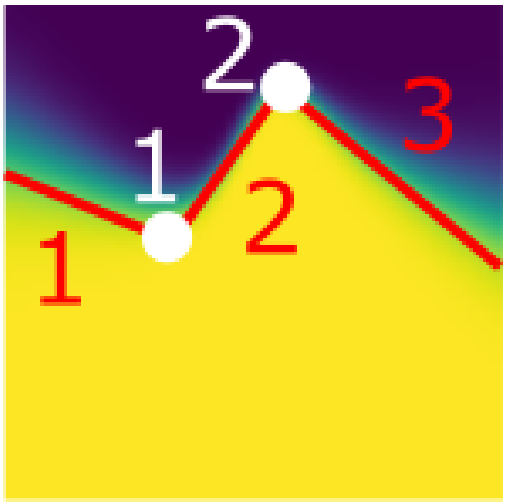


Figure 6-11. Example conversion of neural network to explainable function. A piecewise linear function is fit, with nodes (white) and lines (red). The decision boundary was derived from Figure 6-10, panel 4.

around 2. As a reminder, values of Input 2 were drawn from the range [-1, 1], even though the plotted range of Input 2 is [-2, 2]. Because the component behavior changes rapidly beyond its input range, the component may not "extrapolate" well beyond the range of its training data.

6.7. Converting Neural Networks to Explainable Models

Novel components cannot implement arbitrary neural networks. Therefore, neural network approximations of novel component behavior need to be converted to explainable functions. Here, we demonstrate a method to convert the neural network to an explainable model, and we demonstrate that the process is reasonably robust to approximation error, at least for the problem that we studied.

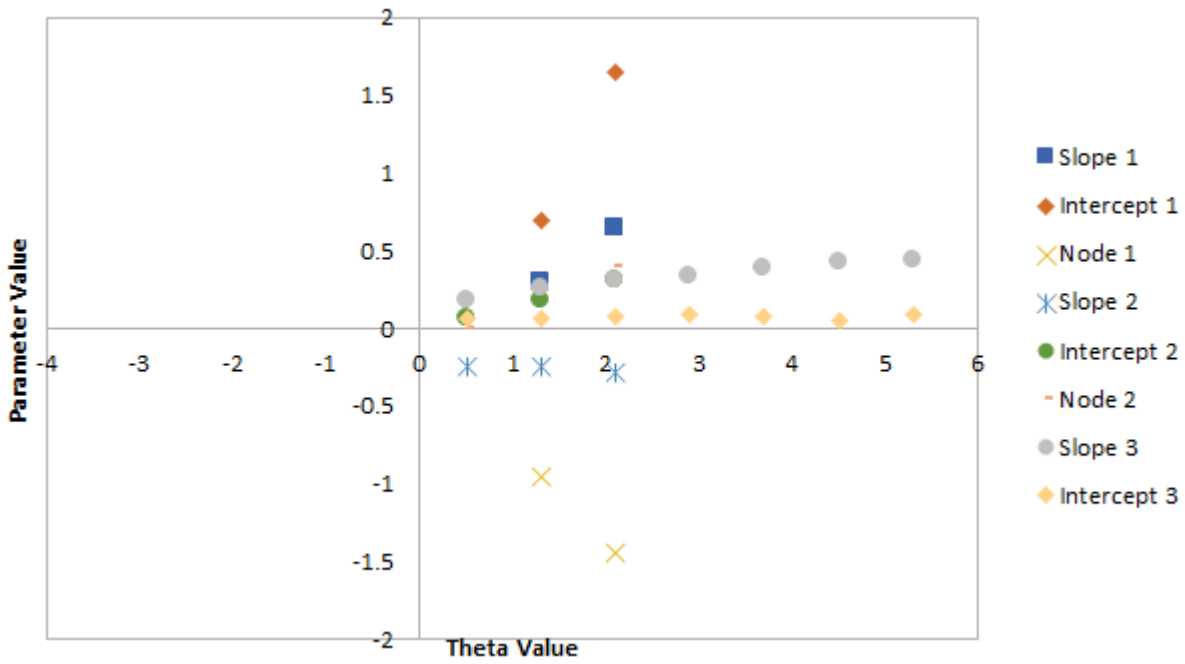


Figure 6-12. Piecewise linear parameters as functions of θ_1

By studying Figure 6-10, it is clear that θ_1 is scaling and shifting the decision area. In order to convert the decision area to an explainable function, we approximated the decision boundary with a piecewise linear function, as shown in Figure 6-11. We fit these lines for multiple values of θ_1 and then plotted line parameters (slope m and intercept b) as well as node parameters (location on x-axis) as functions of θ_1, as shown in Figure 6-12. Conveniently, the parameters that defined the decision boundary could be expressed as simple linear functions of θ_1.

After expressing the piecewise linear decision boundary parameters as functions of θ_1, we can plot the approximate decision boundaries, as shown in Figure 6-13. By comparing Figure 6-13 to Figure 6-10, we see that the piecewise linear fit is a fairly good approximation. However, there are some minor differences. For instance, by comparing the first panel of both figures, it is clear that the piecewise linear fit has a distinct output-0 region that is absent from the neural network approximation. These discrepancies can be explained by the few data points (few values of θ_1) that were used to create the piecewise linear fit. We can also see that the output of the neural network approximation decision boundary involves a relatively gradual transition from 0 to 1, whereas the piecewise linear decision boundary creates an abrupt transitions between outputs of 0 and 1.

We asked an RL agent to build circuits using the piecewise linear approximation of the neural network approximation. The RL-agent needed to specify the value of θ_1. The RL agent was able to choose the piecewise linear approximations of the novel component to perform the task with 100% success. The RL agent didn't need to learn the input-output characteristics of the novel component, it only needed to learn how to use the piecewise linear components. Thus, training was much faster than when the RL agent needed to learn the input-output characteristics of the novel component. Additionally, because we knew the range of θ_1 componentence values that

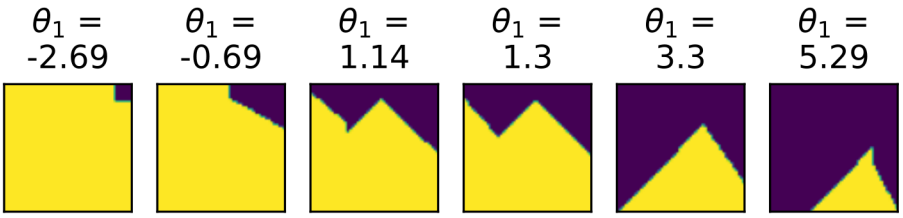


Figure 6-13. Piecewise linear approximation of decision boundaries. Compare to Figure 6-10

were required to solve the problem, we were able to modify the output range of the RL agent to allow it to learn faster - at least qualitatively.

This experiment demonstrated that we can extract useful information from neural network models of novel components using simpler approximations. The simplifying approximations did not seem to affect the performance of circuits that were built with the novel components, at least for this problem of interest.

6.8. Discussion

6.8.1. Summary

We asked the AI-enhanced circuit design tool to design novel components and use those components to perform a signal detection task. The tool was able to perform the task with high accuracy (approaching 100%). By slightly changing the problem statement, we were able to induce the AI-enhanced circuit design tool to produce novel modules with very different input-output behavior. We developed methods to analyze the neural network models of the the novel modules, and our analyses gave us insight, not only into the function of the novel modules, but also into the stated problem. The analysis methods allowed us to develop explainable models of the novel modules. RL was able to arrange the explainable models into circuits that could perform the task with high accuracy, even though the explainable models were noticeably different than the neural network models that were originally optimized for the task.

6.8.2. Future Work

Here, we discovered novel component behavior by using a training process that alternated between training the RL agent (to output componentence values) and training the neural network component model (to better use the componentence values output by the RL agent). This process was quite time consuming. Novel module discovery could be potentially accelerated by making the neural network module models a part of the RL agent. This could potentially alleviate the

need to alternate between training the RL agent and training the neural network component model, which could save time.

The problem that we studied involved a simple detection task that represented the the least complex problem that was still interesting. Importantly, the task that we chose required the RL agent to learn to output componentence parameters, and it required the novel component model to learn to use the componentence parameters. The problem that we selected could not be solved with a single component that had no componentence parameters. Future work should consider more complex problems, such as detection problems where the distributions of target and non-target signals overlap.

In this work, MAMMAL was manually switched into "component discovery mode" by an investigator. The ability to manually switch MAMMAL into discovery mode is attractive for scientists who are interested in developing new devices, components, or circuits. However, future versions of MAMMAL should include a way for MAMMAL to automatically switch into discovery mode when available components in the library are insufficient to perform the desired task.

7. DISCUSSION AND CONCLUSIONS

This work made several key contributions as listed below:

7.1. MAMMAL (Reinforcement Learning)

AI-enhanced codesign has the potential to impact many technical fields by accelerating the design process and by enabling the maintenance of institutional knowledge. By simultaneously considering multiple system levels, AI-enhanced codesign has the potential to revolutionize system design.

This project laid the groundwork for AI-enhanced codesign by demonstrating that reinforcement learning can be used to design circuits from components. We called this RL-based system design tool MAMMAL (Section 2.2). Here, we demonstrated that MAMMAL can build circuits of different lengths by choosing circuit components from a library (Chapter 3). Furthermore, we showed that MAMMAL is capable of producing mixed actions. Trained RL agents were able to choose both discrete components and real-valued parameters (Section 3.1.3). Additionally, we discovered several methods for improving the performance of MAMMAL, such as log-transforming rewards (Section 3.1.2). Notably, MAMMAL was tested on a real-world circuit design task that is related to Sandia's mission.

We recognize that circuit design is a complex field because circuits must do more than perform a specified task – they must also meet specified design constraints. As such, we identified a path towards incorporating such constraints into MAMMAL. We successfully demonstrated that MAMMAL can build circuits while minimizing some objective function, like component cost (Section 3.1.4). These same methods should generalize to other constraints such as size, weight, and power.

Many circuit components, such as capacitors, have time-varying outputs with respect to the inputs because of unobserved internal variables. In order to design circuits from such components, it may be necessary for RL agents to observe multiple samples of a time series simultaneously. Unfortunately, MAMMAL had trouble building circuits when time series inputs were presented simultaneously rather than sample-by-sample (Section 3.1.5). However, we expect that future innovations will enable observations of multiple time series samples. In particular, we are excited to try other neural network architectures, such as recurrent neural networks.

Curriculum learning proved a boon for enhancing MAMMAL. By employing curriculum learning, we were able to massively accelerate design without resorting to dense rewards that are not available in the real world (Section 3.2.3). Curriculum learning allowed us to maintain a linear growth in training time vs. signal length while using sparse rewards. While investigating

RL acceleration methods, we discovered that some circuit design problems benefit from zero-shot transfer learning (see, for instance, Section 3.3.1). Zero-shot transfer learning allowed us to efficiently train MAMMAL on short circuits and then use MAMMAL to design long circuits without further training.

7.2. Evolutionary Algorithms

In addition to reinforcement learning methods, we explored if evolutionary algorithms could be used to rapidly prototype circuits. We hypothesized that evolutionary algorithms could be used to rapidly build circuits by using emerging devices, such as the Mott memristor. To test this hypothesis, we designed a simple evolutionary algorithm that was able to rapidly build signal detection circuits from Mott memristors (Chapter 5). Additionally, we compared MAMMAL and evolutionary algorithms (Chapter 4). Evolutionary algorithms can create one-off circuit designs more quickly, but RL methods can more rapidly design many circuits. Because evolutionary algorithms find solutions to specific problems, rather than families of problems, evolutionary algorithms produce circuit designs that work better on test cases. While higher performance is certainly desirable, it may be necessary to validate evolutionary algorithms to confirm that they do not overfit the data. It may be possible to use evolutionary algorithms to optimize initial RL designs in order to take advantage of the strengths of both methods.

7.3. Novel Device Discovery

The process of discovery is characterized by long periods of incremental development that are punctuated with sudden insight. By stimulating insight, we can expect to accelerate discovery. During this project, we developed methods that enabled MAMMAL to stimulate insight by suggesting specifications for novel devices that can solve known problems. These methods will revolutionize the search for new devices by providing novel, unexplored solutions that may be unintuitive to humans. In future work, we expect that these methods will be extended from devices to materials. In doing so, we expect that MAMMAL will be able to predict, not just the function of a novel device, but also the structure.

7.4. Impact and Future Work

In the next year, we intend to improve MAMMAL to allow for more general use in designing digital circuits, analog circuits, and even computer code. We will also work to improve MAMMAL so that it can perform true codesign (design across multiple system levels). We intend to implement codesign by allowing MAMMAL to design a hierarchy of nested modules at multiple system levels. We also plan to implement neural networks that can enable MAMMAL to design systems that have components with unobserved variables. Through PyMi [22], we will integrate MAMMAL with Xyce [20], an open-source large scale circuit simulator to allow for circuit simulations at higher fidelity. Beyond these design goals, we intend to test how

MAMMAL performs when the size of the component library is increased and when the circuit complexity is increased.

In conclusion, we remark that, as MAMMAL matures, it will disrupt the circuit design process as we know it today. It is critical to continue developing MAMMAL. MAMMAL is the first RL approach to create circuit designs from scratch, and we have found it to be wildly successful in creating functional designs. Similar commercially-available circuit optimization techniques are relegated to microelectronic design sub-tasks, such as chip floorplanning. While, these commercially-available tools have already accelerated microelectronics design, MAMMAL can be expected to further accelerate the process by removing the rate-limiting step: the development of an initial human design. We see natural extensions from our methodology to similar RL methods for engineering design, including parts and assemblies, civil infrastructure, and commercial and industrial processes. We intend to develop similar RL techniques that are applicable in these aforementioned technical fields.

REFERENCES

[1] Mythic AI. https://mythic.ai/, 2022. [Online; accessed 09-September-2022].

[2] Marcin Andrychowicz, Filip Wolski, Alex Ray, Jonas Schneider, Rachel Fong, Peter Welinder, Bob McGrew, Josh Tobin, OpenAI Pieter Abbeel, and Wojciech Zaremba. Hindsight experience replay. *Advances in neural information processing systems*, 30, 2017.

[3] Jakob Axelsson. Game theory applications in systems-of-systems engineering: A literature review and synthesis. *Procedia Computer Science*, 153:154–165, 2019.

[4] Ben Varkey Benjamin, Peiran Gao, Emmett McQuinn, Swadesh Choudhary, Anand R Chandrasekaran, Jean-Marie Bussat, Rodrigo Alvarez-Icaza, John V Arthur, Paul A Merolla, and Kwabena Boahen. Neurogrid: A mixed-analog-digital multichip system for large-scale neural simulations. *Proceedings of the IEEE*, 102(5):699–716, 2014.

[5] Stephen Brink, Stephen Nease, Paul Hasler, Shubha Ramakrishnan, Richard Wunderlich, Arindam Basu, and Brian Degnan. A learning-enabled neuron array ic based upon transistor channel models of biological phenomena. *IEEE Transactions on Biomedical Circuits and Systems*, 7(1):71–81, 2012.

[6] Ahmet F Budak, Prateek Bhansali, Bo Liu, Nan Sun, David Z Pan, and Chandramouli V Kashyap. Dnn-opt: An rl inspired optimization for analog circuit sizing using deep neural networks. In *2021 58th ACM/IEEE Design Automation Conference (DAC)*, pages 1219–1224. IEEE.

[7] Cadence. Cadence cerebrus intelligent chip explorer. https://www.cadence.com/en_US/home/tools/digital-design-and-signoff/soc-implementation-and-floorplanning/cerebrus-intelligent-chip-explorer.html.

[8] Colin F Camerer. Does strategy research need game theory? *Strategic Management Journal*, 12(S2):137–152, 1991.

[9] K Daniel Cooksey and Dimitri Mavris. Game theory as a means of modeling system of systems viability and feasibility. In *2011 Aerospace Conference*, pages 1–11. IEEE, 2011.

[10] Mike Davies, Narayan Srinivasa, Tsung-Han Lin, Gautham Chinya, Yongqiang Cao, Sri Harsha Choday, Georgios Dimou, Prasad Joshi, Nabil Imam, Shweta Jain, et al. Loihi: A neuromorphic manycore processor with on-chip learning. *Ieee Micro*, 38(1):82–99, 2018.

[11] Steve B Furber, Francesco Galluppi, Steve Temple, and Luis A Plana. The spinnaker project. *Proceedings of the IEEE*, 102(5):652–665, 2014.

[12] Timothy Ganesan, I Elamvazuthi, and Pandian Vasant. Multiobjective design optimization of a nano-cmos voltage-controlled oscillator using game theoretic-differential evolution. *Applied Soft Computing*, 32:293–299, 2015.

[13] Suma George, Jennifer Hasler, Scott Koziol, Stephen Nease, and Shubha Ramakrishnan. Low power dendritic computation for wordspotting. *Journal of Low Power Electronics and Applications*, 3(2):73–98, 2013.

[14] Suma George, Sihwan Kim, Sahil Shah, Jennifer Hasler, Michelle Collins, Farhan Adil, Richard Wunderlich, Stephen Nease, and Shubha Ramakrishnan. A programmable and configurable mixed-mode fpaa soc. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 24(6):2253–2261, 2016.

[15] Avner Greif. Economic history and game theory. *Handbook of game theory with economic applications*, 3:1989–2024, 2002.

[16] Tuomas Haarnoja, Aurick Zhou, Pieter Abbeel, and Sergey Levine. Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor. In *International conference on machine learning*, pages 1861–1870. PMLR, 2018.

[17] Sara Hooker. The hardware lottery. *Communications of the ACM*, 64(12):58–65, 2021.

[18] Sebastian Höppner, Yexin Yan, Andreas Dixius, Stefan Scholze, Johannes Partzsch, Marco Stolba, Florian Kelber, Bernhard Vogginger, Felix Neumärker, Georg Ellguth, et al. The spinnaker 2 processing element architecture for hybrid digital neuromorphic computing. *arXiv preprint arXiv:2103.08392*, 2021.

[19] G. Y. Huang, J. B. Hu, Y. F. He, J. L. Liu, M. Y. Ma, Z. Y. Shen, J. J. Wu, Y. F. Xu, H. R. Zhang, K. Zhong, X. F. Ning, Y. Z. Ma, H. Y. Yang, B. Yu, H. Z. Yang, and Y. Wang. Machine learning for electronic design automation: A survey. *Acm Transactions on Design Automation of Electronic Systems*, 26(5), 2021. Huang, Guyue Hu, Jingbo He, Yifan Liu, Jialong Ma, Mingyuan Shen, Zhaoyang Wu, Juejian Xu, Yuanfan Zhang, Hengrui Zhong, Kai Ning, Xuefei Ma, Yuzhe Yang, Haoyu Yu, Bei Yang, Huazhong Wang, Yu 1557-7309.

[20] S Hutchinson, E Keiter, R Hoekstra, H Watts, A Waters, T Russo, R Schells, S Wix, and C Bogdan. The xyce^{TM} parallel electronic simulator–an overview. *Parallel Computing: Advances and Current Issues*, pages 165–172, 2002.

[21] Christof Koch and Idan Segev. The role of single neurons in information processing. *Nature Neuroscience*, 3:1171–1177, November 2000.

[22] Paul Kuberry and Eric Keiter. An embedded python model interpreter for xycetm (xyce-pymi). Technical report, Sandia National Lab.(SNL-NM), Albuquerque, NM (United States), 2021.

[23] Suhas Kumar, R Stanley Williams, and Ziwen Wang. Third-order nanocircuit elements for neuromorphic engineering. *Nature*, 585(7826):518–523, 2020.

[24] Gary Lauterbach. The path to successful wafer-scale integration: The cerebras story. *IEEE Micro*, 41(6):52–57, 2021.

[25] Yiyang Li, T Patrick Xiao, Christopher H Bennett, Erik Isele, Armantas Melianas, Hanbo Tao, Matthew J Marinella, Alberto Salleo, Elliot J Fuller, and A Alec Talin. In situ parallel training of analog neural network using electrochemical random-access memory. *Frontiers in Neuroscience*, 15:636127, 2021.

[26] Michael London and Michael Hausser. Dendritic computation. *Annual Review of Neuroscience*, 28:503–532, July 2005.

[27] Ratul Mahajan, Maya Rodrig, David Wetherall, and John Zahorjan. Experiences applying game theory to system design. In *Proceedings of the ACM SIGCOMM workshop on Practice and theory of incentives in networked systems*, pages 183–190, 2004.

[28] Christian Mayr, Sebastian Hoeppner, and Steve Furber. Spinnaker 2: A 10 million core processor system for brain simulation and machine learning. *arXiv preprint arXiv:1911.02385*, 2019.

[29] Carver Mead. *Analog VLSI and Neural Systems*. Addison-Wesley, Reading, MA, 1989.

[30] Paul A Merolla, John V Arthur, Rodrigo Alvarez-Icaza, Andrew S Cassidy, Jun Sawada, Filipp Akopyan, Bryan L Jackson, Nabil Imam, Chen Guo, Yutaka Nakamura, et al. A million spiking-neuron integrated circuit with a scalable communication network and interface. *Science*, 345(6197):668–673, 2014.

[31] Azalia Mirhoseini, Anna Goldie, Hieu Pham, Benoit Steiner, Quoc V Le, and Jeff Dean. A hierarchical model for device placement. In *International Conference on Learning Representations*, 2018.

[32] Azalia Mirhoseini, Anna Goldie, Mustafa Yazgan, Joe Wenjie Jiang, Ebrahim Songhori, Shen Wang, Young-Joon Lee, Eric Johnson, Omkar Pathak, Azade Nazi, et al. A graph placement methodology for fast chip design. *Nature*, 594(7862):207–212, 2021.

[33] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013.

[34] Saber Moradi, Ning Qiao, Fabio Stefanini, and Giacomo Indiveri. A scalable multicore architecture with heterogeneous memory structures for dynamic neuromorphic asynchronous processors (dynaps). *IEEE transactions on biomedical circuits and systems*, 12(1):106–122, 2017.

[35] Stephen Nease, Suma George, Paul Hasler, Scott Koziol, and Stephen Brink. Modeling and implementation of voltage-mode cmos dendrites on a reconfigurable analog platform. *IEEE transactions on biomedical circuits and systems*, 6(1):76–84, 2011.

[36] Intel Newsroom. *Intel Scales Neuromorphic Research System to 100 Million Neurons*, March 18, 2020. Accessed June 13th, 2020.

[37] Andrew Y Ng, Daishi Harada, and Stuart Russell. Policy invariance under reward transformations: Theory and application to reward shaping. In *Icml*, volume 99, pages 278–287, 1999.

[38] Garrick Orchard, E Paxon Frady, Daniel Ben Dayan Rubin, Sophia Sanborn, Sumit Bam Shrestha, Friedrich T Sommer, and Mike Davies. Efficient neuromorphic signal processing with loihi 2. In *2021 IEEE Workshop on Signal Processing Systems (SiPS)*, pages 254–259. IEEE, 2021.

[39] Ning Qiao and Giacomo Indiveri. Scaling mixed-signal neuromorphic processors to 28 nm fd-soi technologies. In *2016 IEEE Biomedical Circuits and Systems Conference (BioCAS)*, pages 552–555. IEEE, 2016.

[40] Antonin Raffin, Ashley Hill, Adam Gleave, Anssi Kanervisto, Maximilian Ernestus, and Noah Dormann. Stable-baselines3: Reliable reinforcement learning implementations. *Journal of Machine Learning Research*, 22(268):1–8, 2021.

[41] Shubha Ramakrishnan, Richard Wunderlich, Jennifer Hasler, and Suma George. Neuron array with plastic synapses and programmable dendrites. *IEEE transactions on biomedical circuits and systems*, 7(5):631–642, 2013.

[42] Martin Rapp, Hussam Amrouch, Yibo Lin, Bei Yu, David Z Pan, Marilyn Wolf, and Jörg Henkel. Mlcad: A survey of research in machine learning for cad keynote paper. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2021.

[43] H. X. Ren, S. Godil, B. Khailany, R. Kirby, H. G. Liao, S. Nath, J. Raiman, R. Roy, and Ieee. Optimizing vlsi implementation with reinforcement learning - iccad special session paper. In *40th IEEE/ACM International Conference on Computer Aided Design (ICCAD)*, ICCAD-IEEE ACM International Conference on Computer-Aided Design, 2021. Ren, Haoxing Godil, Saad Khailany, Brucek Kirby, Robert Liao, Haiguang Nath, Siddhartha Raiman, Jonathan Roy, Rajarshi Roy, Rajarshi/0000-0003-4548-2114 1933-7760.

[44] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

[45] Catherine D Schuman, J Parker Mitchell, Robert M Patton, Thomas E Potok, and James S Plank. Evolutionary optimization for neuromorphic systems. In *Proceedings of the Neuro-inspired Computational Elements Workshop*, pages 1–9, 2020.

[46] Keertana Settaluri, Ameer Haj-Ali, Qijing Huang, Kourosh Hakhamaneshi, and Borivoje Nikolic. Autockt: Deep reinforcement learning of analog circuit designs. In *2020 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 490–495. IEEE, 2020.

[47] Muhammad Shafique, Theocharis Theocharides, Christos-Savvas Bouganis, Muhammad Abdullah Hanif, Faiq Khalid, Rehan Hafız, and Semeen Rehman. An overview of next-generation architectures for machine learning: Roadmap, opportunities and challenges in the iot era. In *2018 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 827–832. IEEE, 2018.

[48] Andrew M Smith, Jackson R Mayo, Vivian Kammler, Robert C Armstrong, and Yevgeniy Vorobeychik. Using computational game theory to guide verification and security in hardware designs. In *2017 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, pages 110–115. IEEE, 2017.

[49] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT press, 2018.

[50] Synopsis. Dso.ai: Ai-driven design applications. `https://www.synopsys.com/implementation-and-signoff/ml-ai-design/dso-ai.html?cmp=sem_dg_nct_ga_spct_072121_nn_ai_us`.

[51] Thomas L Vincent. Game theory as a design tool. 1983.

[52] John Von Neumann and Oskar Morgenstern. Theory of games and economic behavior, 2nd rev. 1947.

[53] Weier Wan, Rajkumar Kubendran, Clemens Schaefer, Sukru Burc Eryilmaz, Wenqiang Zhang, Dabin Wu, Stephen Deiss, Priyanka Raina, He Qian, Bin Gao, et al. A compute-in-memory chip based on resistive random-access memory. *Nature*, 608(7923):504–512, 2022.

[54] Hanrui Wang, Kuan Wang, Jiacheng Yang, Linxiao Shen, Nan Sun, Hae-Seung Lee, and Song Han. Gcn-rl circuit designer: Transferable transistor sizing with graph neural networks and reinforcement learning. In *2020 57th ACM/IEEE Design Automation Conference (DAC)*, pages 1–6. IEEE, 2020.

[55] Shimeng Yu and Pai-Yu Chen. Emerging memory technologies: Recent trends and prospects. *IEEE Solid-State Circuits Magazine*, 8(2):43–56, 2016.

[56] Dan Zhang, Safeen Huda, Ebrahim Songhori, Kartik Prabhu, Quoc Le, Anna Goldie, and Azalia Mirhoseini. A full-stack search technique for domain optimized deep learning accelerators. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 27–42, 2022.

[57] He Zhang, Yuelong Su, Lihui Peng, and Danya Yao. A review of game theory applications in transportation analysis. In *2010 international conference on computer and information application*, pages 152–157. IEEE, 2010.

[58] Keren Zhu, Mingjie Liu, Hao Chen, Zheng Zhao, and David Z Pan. Exploring logic optimizations with reinforcement learning and graph convolutional network. in 2020 acm/ieee 2nd workshop on machine learning for cad (mlcad), 2020.

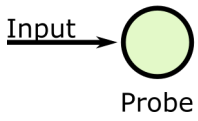
[59] Quanyan Zhu and Stefan Rass. Game theory meets network security: A tutorial. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 2163–2165, 2018.

APPENDIX A. Component Diagrams

A.1. Delay Components

Probe

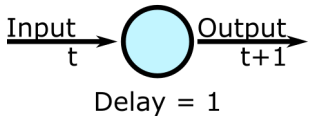
Parameters: None



Terminate the circuit

Delay1

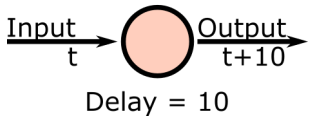
Parameters: None



Delay the input by 1 time step

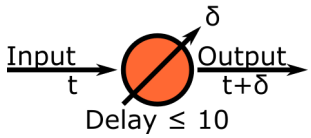
Delay10

Parameters: None



Delay the input by 10 time steps

Delta Delay

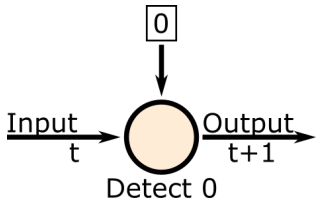


Parameters: Delay: δ (Real)

Delay the input by δ time steps

A.2. Detect Components

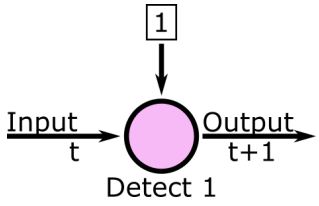
Detect0



Parameters: None

If the 2nd input (top) is a 0, output the 1st input (left) delayed by one time step. Otherwise, output 0.

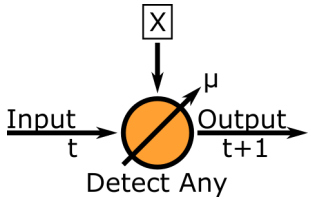
Detect1



Parameters: None

If the 2nd input (top) is a 1, output the 1st input (left) delayed by one time step. Otherwise, output 0

DetectAny

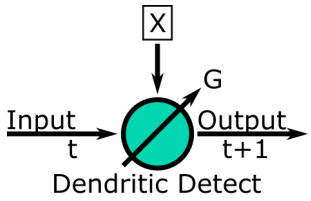


Parameters: Mean of detection distribution: μ

If the 2nd input (top) is within $\pm$ 0.11 of the RL-specified mean μ, output the 1st input (left) delayed by one time step. Otherwise, output 0.

A.3. Dendritic Components

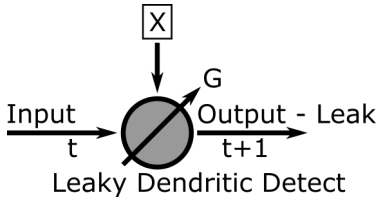
Dendritic Detect



Parameters: Gain: G

Multiply the 2nd input (top) by the Gain, G, and add it to the 1st input (left).

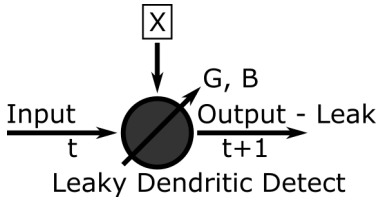
Leaky Dendritic Detect



Parameters: Gain: G

Multiply the 2nd input (top) by the Gain, G, and add it to the 1st input (left). 0.1 units of "charge" leak from the output before being sent to the input of the next component/compartment

Leaky Dendritic Detect w/ Bias

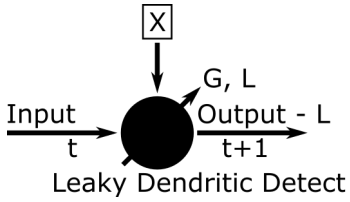


Parameters: Gain: G; Bias: B

Add the bias, B, to the 2nd input (top), and multiply the result by the Gain, G. Add the result to the 1st input (left). 0.1 units of "charge" leak from the output before being sent to the input of the next component/compartment

$Y_{t+1} = G * (X + B) + Y_t - L$

Tunable Leaky Dendritic Detect

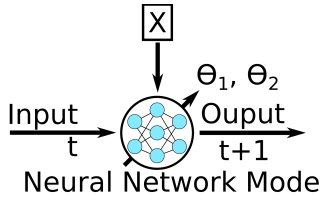


Parameters: Gain: G; Leak: L

Multiply the 2nd input (top) by the Gain, G. Add the result to the 1st input (left), and subtract the leak, L.

$Y_{t+1} = G * X + Y_t - L$

Neural Network Model



Parameters: Parameter: θ_1; Parameter: θ_2

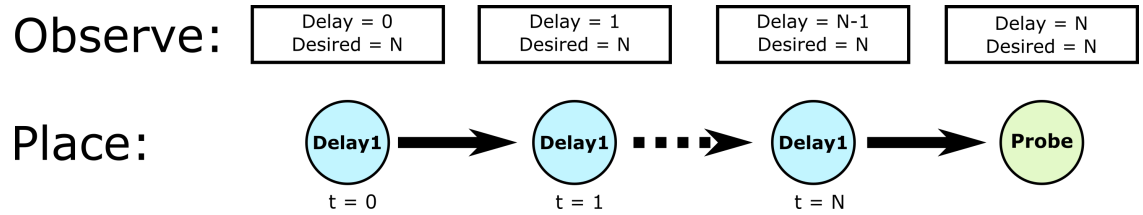
Create a neural network that accepts inputs and 2 free parameters θ_1 and θ_2. The neural network must use the parameters to customize the component behavior since every neural network model component is constrained to have the same internal neural network parameters (e.g., biases and weights).

APPENDIX B. Summaries of RL experiments

Below, we provide concise descriptions of each experiment. Please refer to Chapter 2 for details.

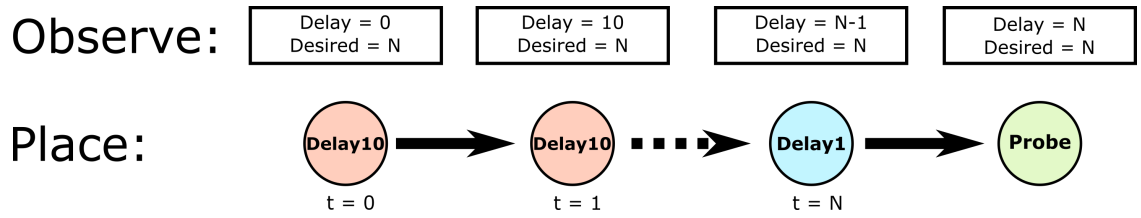
B.1. Delay Experiments

Experiment 1: Simple Delay Line



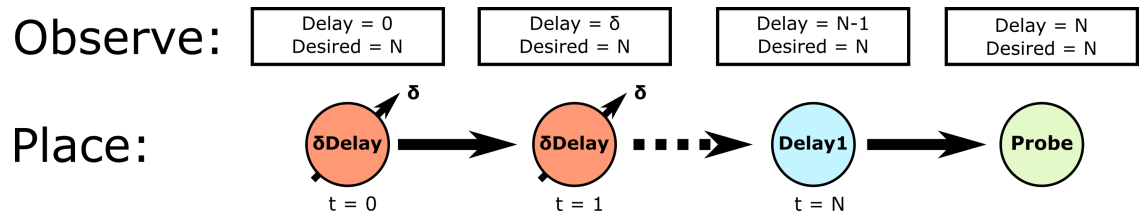
Purpose	Create the simplest proof-of-concept for RL-enhanced circuit design.		
Summary	At each time step, an RL agent is given information about the current delay and the desired delay. The desired delay varies from 0-5 time steps. The RL agent places delay components that delay the input by one time step or places a probe component, which terminates the circuit design process. If no probe component is placed, the design process terminates after 10 time steps.		
Observations	Current Delay	Range	[0, 5]
		Type	Integer
		Channels	1
	Desired Delay	Range	[0, 5]
		Type	Integer
		Channels	1
Rewards	Delay Difference (Δ)	Value	$-1 * \Delta$
		Type	Sparse
Actions	Delay1	Delay input by 1 time step	
	Probe	Terminate circuit and circuit design process	
Algorithm	HER-DQN		
Max # Components	5		
Results	Circuits were created with nearly 100% delay accuracy.		
Insights	RL can place components for circuits of variable lengths.		

Experiment 2: Delay Line w/ 2 Delay Types



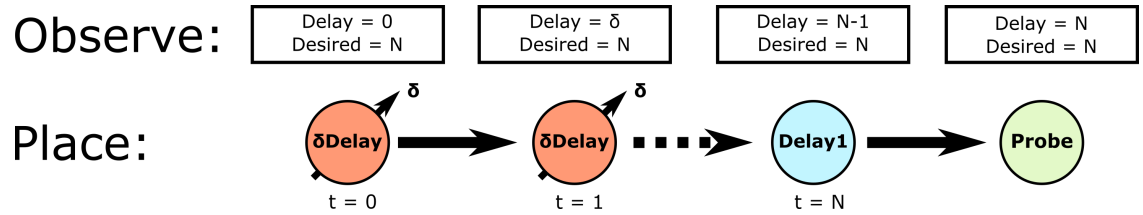
Purpose	Demonstrate that RL can be used to choose components from a library that contains > 2 components.		
Summary	At each time step, an RL agent observes the current delay and the desired delay. The desired delay is in the range [0, 100] time steps. If no probe component is placed (to terminate the design process), the design process terminates after 25 time steps. Note that most delays can only be created by using a combination of Delay1 and Delay10 components		
Observations	Current Delay	Range	[0, 100]
		Type	Integer
		Channels	1
	Desired Delay	Range	[0, 100]
		Type	Integer
		Channels	1
Rewards	Delay Difference (Δ)	Value	$1 - \log_{10}(\Delta)/2$
		Type	Sparse
	Over-Delay	Value	$0.95 \times \Delta$
		Type	Sparse
Actions	Delay1	Delay input by 1 time step	
	Delay10	Delay input by 10 time step	
	Probe	Terminate circuit and circuit design process	
Algorithm	HER-SAC		
Max # Components	10		
Results	Circuits were created with nearly 100% delay accuracy.		
Insights	RL can place ≥ 2 components using continuous actions. Log-transforming rewards is helpful so that RL equally weights Delay1 and Delay10 errors. There is no "backspace" component, which means that placing too many delays is worse than placing too few delays because placing too many delays cannot be fixed.		

Experiment 3: Delay Line w/ Continuous Parameters



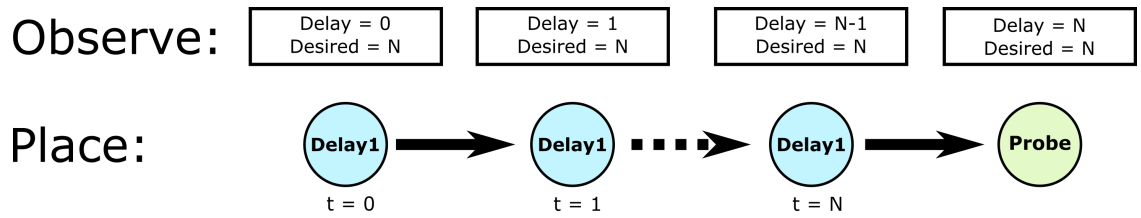
Purpose	Demonstrate that RL can choose components and also set continuous parameter values.		
Summary	At each time step, an RL agent is given information about the current delay and the desired delay. The desired delay varies from 0-100 time steps. The RL agent can place 1) a delay component that delays the input by 1 time step 2) a delay component that delays the input by 0-10 (real-valued) time steps or 3) a probe component that terminates the circuit design process. If no probe component is placed, the design process terminates after 25 time steps.		
Observations	Current Delay	Range	[0, 100]
		Type	Integer
		Channels	1
	Desired Delay	Range	[0, 100]
		Type	Integer
		Channels	1
Rewards	Delay Difference (Δ)	Value	$1 - \log_{10}(\Delta)/2$
		Type	Sparse
	Over-Delay	Value	$0.95 \times \Delta$
		Type	Sparse
Actions	Delay1	Delay input by 1 time units	
	Delta Delay	Delay input by 0-10 (real-valued) time units	
	Probe	Terminate circuit and circuit design process	
Algorithm	HER-SAC		
Max # Components	10		
Results	Circuits were created with nearly 100% delay accuracy.		
Insights	RL can simultaneously place components and set continuous component parameters		

Experiment 4: Delay Line w/ Continuous Parameters & Costs



Purpose	Demonstrate that RL can choose components, which have costs.		
Summary	At each time step, an RL agent is given information about the current delay and the desired delay. The desired delay varies from 0-20 time steps. The RL agent can place 1) a delay component that delays the input by 1 time step 2) a delay component that delays the input by 0-10 time steps or 3) a probe component which terminates the circuit design process. If no probe component is placed, the design process terminates after 25 time steps. This experiment is similar to the Delay Line w/ Continuous Parameters experiment, but components have costs.		
Observations	Current Delay	Range	[0, 20]
		Type	Integer
		Channels	1
	Desired Delay	Range	[0, 20]
		Type	Integer
		Channels	1
Rewards	Delay Difference (Δ)	Value	$1 - \log_{10}(\Delta)/2$
		Type	Sparse
	Over-Delay	Value	$0.95 \times \Delta$
		Type	Sparse
	Delay1 Cost	Value	$-1^{-5} \times$#Delay1
		Type	Sparse
	Delay10 Cost	Value	$-100^{-5} \times$#Delay10
		Type	Sparse
Actions	Delay1	Delay input by 1 time units	
	Delta Delay	Delay input by 0-10 (real-valued) time units	
	Probe	Terminate circuit and circuit design process	
Algorithm	HER-SAC		
Max # Components	10		
Results	Circuits were created with nearly 100% delay accuracy. Qualitatively, costs appeared to be minimized.		
Insights	RL can place components w/ continuous component parameters while minimizing costs.		

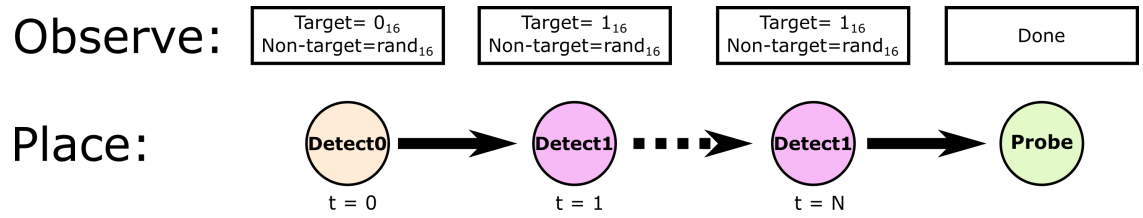
Experiment 5: Delay Line w/ Timeseries Input



Purpose	Components can affect outputs at multiple time steps. Circuit design tools should be able to identify temporal patterns produced by components. In order to identify temporal patterns, circuit design tools need to represent inputs/outputs at multiple timesteps.		
Summary	At every time step, the RL agent observes every time step of the input signal, as well as the current time step number. The desired delay is provided in the form of a delta function. The RL agent can place 1) a delay component that delays the input by 1 time step or 2) a probe component that terminates the circuit design process. If no probe component is placed, the design process terminates after 10 time steps.		
Observations	Current Delay	Range Type Channels	[0, 5] Vector of Integers 1
	Desired Delay	Range Type Channels	[0, 5] Vector of Integers 1
	Current Time Step	Range Type Channels	[0, 10] Integer 1
Rewards	Delay Difference (Δ)	Value Type	$1 - \log_{10}(\Delta)/2$ Sparse
Actions	Delay1 Probe	Delay input by 1 time units Terminate circuit and circuit design process	
Algorithm	HER-SAC		
Max # Components	10		
Results	Training was not successful.		
Insights	RL has trouble learning which component of a vector input should be considered at a given time step. Further work is needed to determine how to efficiently teach an RL agent this skill.		

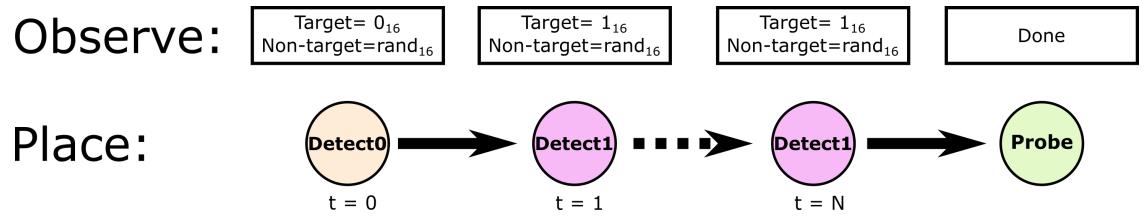
B.2. Detect Experiments

Experiment 6: Simple Delay Gate - Dense Rewards



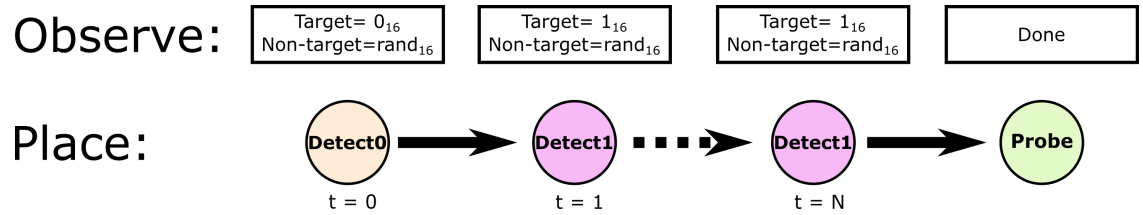
Purpose	Produce a minimum working example for an actual neuromorphic circuit design problem.		
Summary	At each time step, an RL agent is shown 16 example of target signals and 16 examples of a non-target signals. The agent places one of the following components: 1) a Detect1 component that detects a 1, 2) a Detect0 component that detects a 0, 3) a probe component that terminates the circuit design process. If no probe component is placed, the design process terminates after N + 1 time steps, for integer $N \in [0, 10]$. The job of the RL agent is to build circuits to detect target signals. The Digital signal detection task was used.		
Observations	Target Signal	Range	[0, 1]
		Type	Binary
		Channels	16
	Non-target Signal	Range	[0, 1]
		Type	Binary
		Channels	16
Rewards	Correct Component	Value	1
		Type	Dense
Actions	Detect1	Detect an input of 1	
	Detect0	Detect an input of 0	
	Probe	Terminate circuit and circuit design process	
Algorithm	PPO (or DQN)		
Max # Components	0-10		
Results	Detection accuracy approached 100% when using dense rewards (rewards given after each component was placed). Training times increased linearly with signal length.		
Insights	RL can learn to place components by looking at vector-valued (multiple channel) inputs and outputs RL can learn to place neuromorphic components when given dense rewards. However, to be more useful, rewards should be sparse (given only at the end of circuit design). RL can learn to build neuromorphic circuits to discriminate between target and non-target signals.		

Experiment 7: Simple Delay Gate - Sparse Rewards



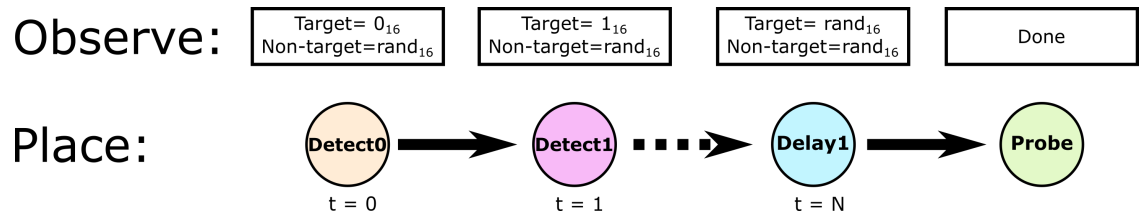
Purpose	Produce a minimum working example for an actual neuromorphic circuit design problem with sparse rewards.		
Summary	At each time step, an RL agent is shown 16 example of target signals and 16 examples of a non-target signals. The agent places one of the following components: 1) a Detect1 component that detects a 1, 2) a Detect0 component that detects a 0, 3) a probe component that terminates the circuit design process. If no probe component is placed, the design process terminates after N + 1 time steps, for integer $N \in [0, 10]$. The job of the RL agent is to build circuits to detect target signals. The Digital signal detection task was used. This is similar to the previous experiment, but with sparse rewards.		
Observations	Target Signal	Range	[0, 1]
		Type	Binary
		Channels	16
	Non-target Signal	Range	[0, 1]
		Type	Binary
		Channels	16
Rewards	Correct Detection	Value	1
		Type	Sparse
Actions	Detect1	Detect an input of 1	
	Detect0	Detect an input of 0	
	Probe	Terminate circuit and circuit design process	
Algorithm	PPO (or DQN)		
Max # Components	0-10		
Results	Detection accuracy approached 100% when using sparse rewards (rewards given only after each circuit was completed) as long as signal length was less than 5. For signal lengths greater than 5, the RL algorithm did not converge. Training times increased exponentially.		
Insights	RL can use sparse rewards to learn to place neuromorphic components for signal lengths less than 5. For signal lengths greater than 5, another method will need to be developed to make training times linear with signal length. Sparse rewards should be used because, for interesting problems, we won't know if each placed component is "correct", but we will know if the circuit output is correct.		

Experiment 8: Simple Delay Gate - Curriculum Learning



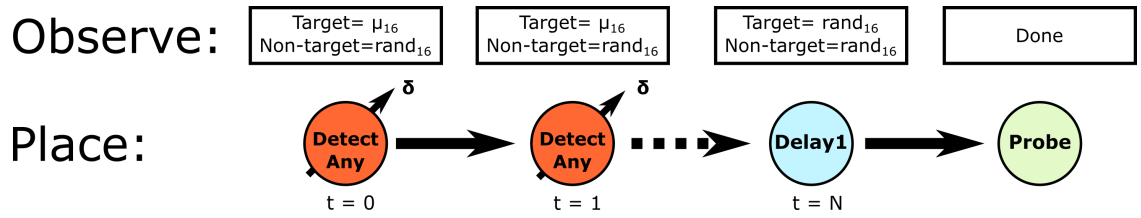
Purpose	Train RL agents under realistic conditions (sparse rewards) in linear time (with respect to signal length).		
Summary	At each time step, an RL agent is shown 16 example of target signals and 16 examples of a non-target signals. The agent places one of the following components: 1) a Detect1 component that detects a 1, 2) a Detect0 component that detects a 0, 3) a probe component that terminates the circuit design process. If no probe component is placed, the design process terminates after N + 1 time steps, for integer $N \in [0, 10]$. The job of the RL agent is to build circuits to detect target signals. The Digital signal detection task was used. This is similar to the previous experiment, but with curriculum learning. In curriculum learning, RL agents first learn to construct circuits for short signal lengths and then construct circuits for long signal lengths		
Observations	Target Signal	Range	[0, 1]
		Type	Binary
		Channels	16
	Non-target Signal	Range	[0, 1]
		Type	Binary
		Channels	16
Rewards	Correct Detection	Value	1
		Type	Sparse
Actions	Detect1	Detect an input of 1	
	Detect0	Detect an input of 0	
	Probe	Terminate circuit and circuit design process	
Algorithm	PPO (or DQN)		
Max # Components	0-10		
Results	Detection accuracy approached 100% when using sparse rewards (rewards given after each component placed) for all tested signal lengths. Learning was slower than when using dense rewards, but still linear (with respect to signal length)		
Insights	Curriculum learning allows for RL agents to learn to build neuromorphic circuits in linear time (with respect to signal length) while still using sparse rewards.		

Experiment 9: Delay Gates with Variable Delays



Purpose	Demonstrate that RL agents can be trained to discriminate between target and non-target signals, even when not every sample is linearly separable.		
Summary	At each time step, an RL agent is shown 16 example of target signals and 16 examples of a non-target signals. The agent places one of the following components: 1) a Detect1 component that detects a 1, 2) a Detect0 component that detects a 0, 3) a Delay component that detects nothing, or 4) a probe component that terminates the circuit design process. If no probe component is placed, the design process terminates after N + 1 time steps, for integer $N \in [0, 10]$. The job of the RL agent is to build circuits to detect target signals. The Digital signal detection task was used.		
Observations	Target Signal	Range	[0, 1]
		Type	Binary
		Channels	16
	Non-target Signal	Range	[0, 1]
		Type	Binary
		Channels	16
Rewards	Correct Detection	Value	1
		Type	Sparse
Actions	Detect1	Detect an input of 1	
	Detect0	Detect an input of 0	
	Delay1	Do not detect anything	
	Probe	Terminate circuit and circuit design process	
Algorithm	PPO (or DQN)		
Max # Components	0-10		
Results	Signal detection accuracy approached 100%.		
Insights	Transfer learning allows circuits to be designed quickly. RL can choose components from libraries that contain as many as 4 components. Future experiments should test if current methods work on larger libraries.		

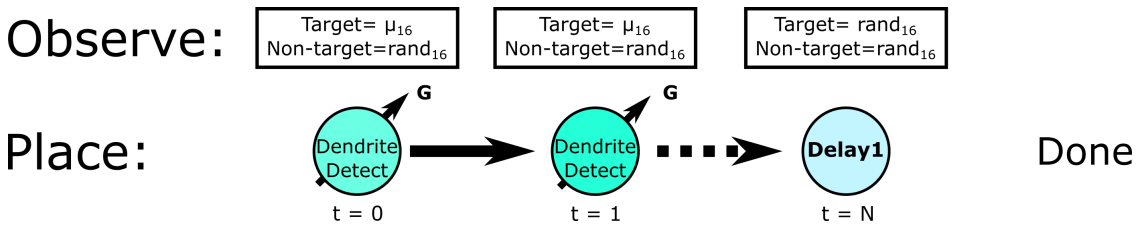
Experiment 10: Delay Gates with Analog Detectors



Purpose	Demonstrate that RL agents can be trained to discriminate between real-valued target and non-target signals.		
Summary	At each time step, an RL agent is shown 16 example of target signals and 16 examples of a non-target signals. The agent places one of the following components: 1) a detect component that detects a signal with a mean chosen by the RL agent and a range of 0.11, 2) a simple delay component, that detects nothing, and 3) a probe component which terminates the circuit design process. If no probe component is placed, the design process terminates after N + 1 time steps, for integer $N \in [0, 20]$. The job of the RL agent is to build circuits to detect target signals. The Low SNR signal detection task was used.		
Observations	Target Signal	Range	[-1, 1]
		Type	Real
		Channels	16
	Non-target Signal	Range	[-1, 1]
		Type	Real
		Channels	16
Rewards	Correct Detection	Value	(#Correct - #Incorrect) / #Total
		Type	Sparse
Actions	DetectAny	Detect an input that is within 0.11 units of the RL-chosen mean	
	Delay1	Do not detect anything	
	Probe	Terminate circuit and circuit design process	
Algorithm	PPO		
Max # Components	0-20		
Results	Signal detection accuracy exceeded 60%. Curriculum learning allowed for signal lengths of 10-20.		
Insights	RL can learn to build neuromorphic circuits to discriminate between target and non-target real-valued signals.		

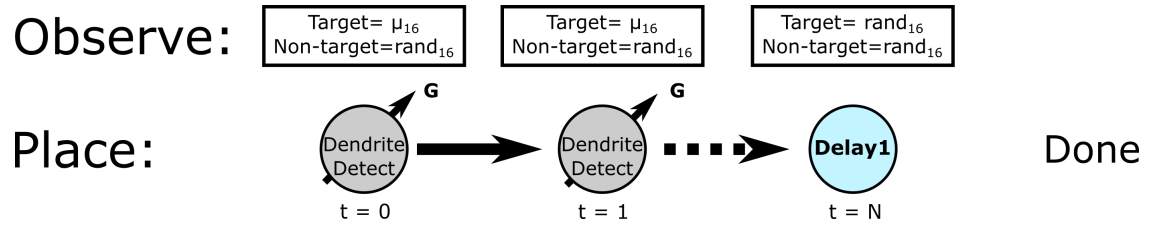
B.3. Dendritic Detect experiments

Experiment 11: Dendritic Delay Line (No Leaks)



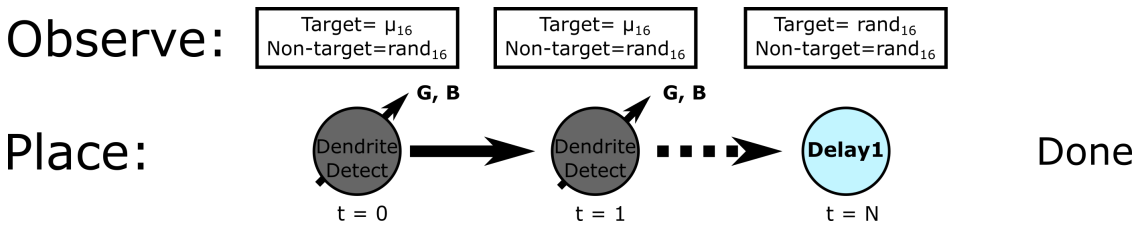
Purpose	Demonstrate that RL agents can be trained to perform detection tasks by using existing neuromorphic hardware.		
Summary	At each time step, an RL agent is shown 16 example of target signals and 16 examples of non-target signals. The agent places one of the following components: 1) a dendritic detect component that amplifies a sample by an RL-chosen gain and adds it to the previous signal or 2) a dendritic delay component that passes the previous signal through with no changes. The job of the RL algorithm is to detect target, but not non-target signals, as described in the previous experiment. The Medium SNR task was used.		
Observations	Target Signal	Range	[-1, 1]
		Type	Real
		Channels	16
	Non-target Signal	Range	[-1, 1]
		Type	Real
		Channels	16
Rewards	Correct Detection	Value	(#Correct - #Incorrect) / #Total
		Type	Sparse
Actions	Dendritic Detect	Multiply the input sample by an RL-chosen gain and add it to the previous signal	
	Delay1	Pass the previous signal through unchanged	
Algorithm	PPO		
Max # Components	10-40		
Results	Signal detection accuracy was approximately 91%.		
Insights	RL can learn to use existing neuromorphic hardware to build circuits to detect signals. Detection tasks exhibit higher performance when target signals are linearly separable from non-target signals. Detection appears to be most accuracte when target and non-target samples have opposite signs. This highlights the benefit of having neuromorphic hardware with an additional bias parameter in each dendritic compartment. For this task, slightly longer signals are required for convergence, possibly so that the RL agent observes a good distribution of each component type. It also seems helpful to balance the proportions of each component type.		

Experiment 12: Dendritic Delay Line (Leaky)



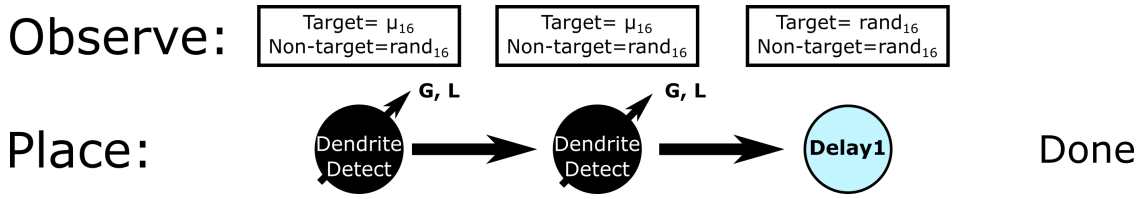
Purpose	Demonstrate that RL agents can be trained to perform detection tasks by using biologically-inspired neuromorphic hardware.		
Summary	At each time step, an RL agent is shown 16 example of target signals and 16 examples of non-target signals. The agent places one of the following components: 1) a dendritic detect component that amplifies a sample by an RL-chosen gain and adds it to the previous signal or 2) a dendritic delay component that passes the previous signal through with no changes. The job of the RL algorithm is to detect target, but not non-target signals, as described in the previous experiment. The Medium SNR task was used. Unlike in the previous non-leaky experiment, 0.1 "charge" leaks from the dendritic compartment before "charge" transfers to the next compartment.		
Observations	Target Signal	Range	[-1, 1]
		Type	Real
		Channels	16
	Non-target Signal	Range	[-1, 1]
		Type	Real
		Channels	16
Rewards	Correct Detection	Value	(#Correct - #Incorrect) / #Total
		Type	Sparse
Actions	Leaky Dendritic Detect	Multiply the input sample by an RL-chosen gain and add it to the previous signal	
	Delay1	Pass the previous signal through unchanged	
Algorithm	PPO		
Max # Components	10-40		
Results	Signal detection accuracy was approximately 91%.		
Insights	RL can learn to use existing neuromorphic hardware to build circuits to detect signals.		

Experiment 13: Dendritic Delay Line (Leaky) w/ Bias



Purpose	Demonstrate that AI-enhanced circuit design tools can be used to rapidly iterate on designs.		
Summary	At each time step, an RL agent is shown 16 example of target signals and 16 examples of non-target signals. The agent places one of the following components: 1) a dendritic detect component that amplifies a sample by an RL-chosen gain and adds it to the previous signal or 2) a dendritic delay component that passes the previous signal through with no changes. The job of the RL algorithm is to detect target, but not non-target signals, as described in the previous experiment. The Medium SNR task was used. Unlike in the previous non-leaky experiment, 0.1 "charge" leaks from the dendritic compartment before "charge" transfers to the next compartment. Unlike in the previous experiment, components had a tunable bias term.		
Observations	Target Signal	Range	[-1, 1]
		Type	Real
		Channels	16
	Non-target Signal	Range	[-1, 1]
		Type	Real
		Channels	16
Rewards	Correct Detection	Value	(#Correct - #Incorrect) / #Total
		Type	Sparse
Actions	Biased Leaky Dendritic Detect	Add an RL-chosen bias to the sample, multiply the result by an RL-chosen gain, and add it to the previous signal	
	Delay1	Pass the previous signal through unchanged	
Algorithm	PPO		
Max # Components	10-40		
Results	Previous experiments were completed with the bias term set to 0. When the bias term was allowed to be learned, signal detection accuracy increased from 91% to 98%.		
Insights	A bias term can improve the ability of neuromorphic circuits to detect target signals. RL can be used to place components and optimize at least 2 parameters. AI-enhanced circuit design tools allow for rapid iteration on circuit designs.		

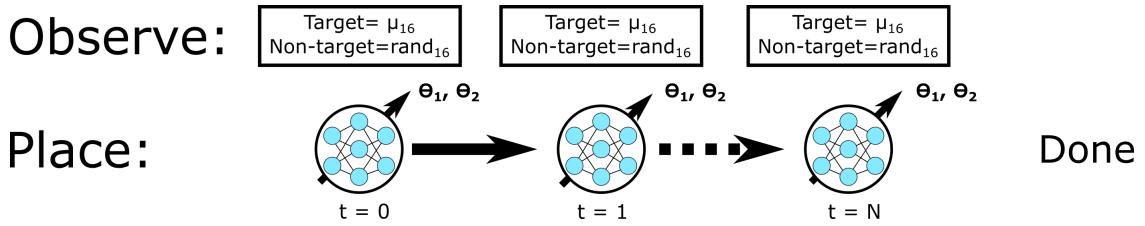
Experiment 14: Dendritic Delay Line (Leaky) w/ Tunable Leak



Purpose	Demonstrate that AI-enhanced circuit design tools can be used to prototype new innovations, even if they do not work.		
Summary	At each time step, an RL agent is shown 16 example of target signals and 16 examples of non-target signals. The agent places one of the following components: 1) a dendritic detect component that amplifies a sample by an RL-chosen gain and adds it to the previous signal or 2) a dendritic delay component that passes the previous signal through with no changes. The job of the RL algorithm is to detect target, but not non-target signals, as described in the previous experiment. The Medium SNR task was used. Unlike in the previous non-leaky experiment, 0.1 "charge" leaks from the dendritic compartment before "charge" transfers to the next compartment. Unlike in the previous experiment, components had a tunable leak term.		
Observations	Target Signal	Range	[-1, 1]
		Type	Real
		Channels	16
	Non-target Signal	Range	[-1, 1]
		Type	Real
		Channels	16
Rewards	Correct Detection	Value	(#Correct - #Incorrect) / #Total
		Type	Sparse
Actions	Tunable Leaky Dendritic Detect	Multiply the sample by an RL-chosen gain, and add it to the previous signal. Subtract a tunable leak.	
	Delay1	Pass the previous signal through unchanged	
Algorithm	PPO		
Max # Components	10-40		
Results	Previous experiments were completed with the leak term set to a constant 0.1. When the leakage term was allowed to be learned, signal detection accuracy did not improve.		
Insights	AI-enhanced circuit design tools allow for rapid iteration on circuit designs, even when designs do not perform as desired.		

B.4. Novel Device Discovery Experiments

Experiment 15: Neural Networks for Novel Components



Purpose	Demonstrate that AI-enhanced tools can produce specifications for novel devices that can solve the given problem.		
Summary	At each time step, an RL agent is shown 16 example of target signals and 16 examples of non-target signals. The agent places a neural network component, which has 2 tunable parameters. The job of the RL algorithm is to detect target, but not non-target signals, as described in the previous experiment. Unlike in the previous experiment, components are neural networks. The RL agent must tune the component input parameters θ_1 and θ_2 in order to perform the task. The Separable About Zero task was used to allow for easy interpretation.		
Observations	Target Signal	Range	[-1, 1]
		Type	Real
		Channels	16
	Non-target Signal	Range	[-1, 1]
		Type	Real
		Channels	16
Rewards	Correct Detection	Value	(#Correct - #Incorrect) / #Total
		Type	Sparse
Actions	Neural Network Model	Train neural network components to perform the necessary task.	
Algorithm	PPO		
Max # Components	10-40		
Results	AI-enhanced tools were able to tune module input parameters (and component models) to perform the task with nearly 100% success.		
Insights	AI-enhanced novel device discovery tools can produce neural networks that represent specifications for novel devices that can solve identified problems.		

DISTRIBUTION

Hardcopy—Internal

Number of Copies	Name	Org.	Mailstop
1	Legal Intellectual Property	11500	0161
1	L. Martin, LDRD Office	1910	0359

Email—Internal (encrypt for OUO)

Name	Org.	Sandia Email Address
Technical Library	1911	sanddocs@sandia.gov

Sandia National Laboratories is a multimission laboratory managed and operated by National Technology & Engineering Solutions of Sandia LLC, a wholly owned subsidiary of Honeywell International Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525.