

Adaptive Evolution and Deployment of Neuromorphic Agents

A Thesis Presented for the

Master of Science

Degree

The University of Tennessee, Knoxville

Ethan Hunter Maness

December 2025

© by Ethan Hunter Maness, 2025
All Rights Reserved.

Acknowledgements

I would like to thank the University of Tennessee-Oak Ridge Innovation Institute for providing the funding to make this work possible. Although this began as an exploration in engine control systems, it has expanded far beyond that initial scope, which could not have happened without their support.

I would like to thank my fellow collaborators at the University of Tennessee. Karan Patel provided much insight in the design of this project and elegantly handled the development of computer engineering tasks I do not have the skills to so easily tackle. Tyler Nitzsche single-handedly created the component of this work that allowed it to actually function on a real system and kept it up to date as we tested it with new issues and problems.

I would like to thank the Oak Ridge National Lab team, consisting of Bryan Maldonado, Brian Kaul, Brett Witherspoon, and Aaron Young, without whom this project could never have been more than a simple software demonstration. Not only did they build a hardware and firmware system to interface between our work and an expensive industrial combustion engine, but they campaigned to give us as much time as possible to test it on the engine.

Lastly, but most importantly, I would like to thank my advisor, Dr. Schuman, for supporting and guiding me throughout this process, and for inspiring me to pursue a graduate degree at all. Her passion and compassion have affected my journey in ways I will never fully know.

Abstract

Neuromorphic computing systems are attractive for real-time control at the edge because of their low power operation, real-time processing capabilities and their potential for rapid online learning. In this work, we describe NODES, a framework for performing real-time evolution of spiking neural networks for neuromorphic systems at the edge. This framework enables the generation and deployment of networks specializing in solving the desired application as it exists under the current real-world conditions by reverse-engineering those conditions and replicating them in simulation for training. We apply this approach to a pole-balancing application as well as a real-time combustion engine control application and examine the exhibited behaviors and what they indicate about the capabilities and limitations of this framework.

Table of Contents

1	Introduction	1
1.1	Neuromorphic Computing	2
1.2	The Simulation-to-Reality Gap	4
1.3	Our Solution	5
2	Related Work	7
3	Methods	9
3.1	Spiking Neural Networks	9
3.2	EONS	10
3.3	TENNLab Neuromorphic Framework	12
3.4	NODES	13
3.4.1	Training	13
3.4.2	Parameter Optimization	16
3.4.3	Interfacing with Hardware	17
3.4.4	The Coordinator	18
4	Case Study 1: Polebalance	23
4.1	Simulator Details	24
4.2	Results	28
5	Case Study 2: Engine Control	31
5.1	Problem Specifics	33

5.2	Engine Modeling	34
5.3	Hardware Infrastructure	37
5.3.1	Hardware Infrastructure Performance	39
5.4	Results	41
6	Conclusion	45
	Bibliography	48
	Vita	51

List of Figures

1.1	Example of a small SNN with an organic architecture. Input neurons are gray, output neurons are blue (bottom), and hidden neurons are purple (top). Synapse direction is indicated by shading, with spikes traveling from light to dark. Note that input and output neurons can both emit and receive spikes, allowing them to participate in computing rather than only encoding and decoding.	3
3.1	Simplified flowchart of NODES processes.	14
3.2	Sim Scheduler runs multiple simulators in parallel.	15
3.3	Flowchart of how the Sim Modifier process approximates matching simulation parameters from real-world data.	17
3.4	The naive method of implementing the Leaderboard. Each dot represents an evaluation of a network, with the position of the dot indicating the simulation parameters used for evaluation. The red circle visualizes a radial search range of parameter values around the approximated real-world values, represented by the red dot.	20
3.5	The binned approach to the Leaderboard. Gray lines delineate each bin. Note that although the search region contains many entries, only one needs to be checked in each bin.	22

4.1	Mean scores of collections of high-scoring networks trained on Polebalance with specific params on a variety of different parameters. Note the trend of scores decreasing as the parameter values move further from the training values.	26
4.2	NODES is run for roughly 2 hours and 20 minutes on the Polebalance application. The Leaderboard is not populated with any pretrained networks except for the single necessary startup network. The blue line shows the average per-episode reward of the preceding two minutes over the duration of the test. Dotted vertical lines indicate when the parameters were manually changed, and red dots indicate when the deployed network was swapped.	29
4.3	NODES is run for roughly 1 hour and 20 minutes on the Polebalance application. The Leaderboard is populated with 121 pretrained networks so that for all possible parameters configurations, it contains one or more networks that perform well. Black dots represent a network from the pretrained batch being deployed.	30
5.1	Cummins ISB 6.7L Engine	32
5.2	Comparison between experimental and simulated values for the physics-based model at 7.4 mg of diesel quantity and 42 deg dTDC injection timing.	35
5.3	Experimental conditions at an average NH_3 replacement rate of 78.9%.	36
5.4	The FireBox hardware configured for co-simulation with hardware-in-the-loop and software diesel engine simulator.	38
5.5	A diagram of the hardware and software services implemented on the FireBox neuromorphic engine controller.	40
5.6	Average network loading time and inference time for various trained networks on the FireBox neuromorphic engine controller.	41

5.7	Plot of cost over a 30-minute interval calculated from observations on the physical engine while being controlled by NODES. Annotations indicate observation, action, and parameter values.	43
-----	---	----

Chapter 1

Introduction

Machine Learning is no longer a novel, burgeoning field of study. Over the course of the past fifteen years, the realm of technologies involving complex autonomous agents has exploded dramatically in breadth and depth while beginning to stabilize into something resembling a state of maturity. Trained models now excel at tasks considered intractable only a decade ago. Though the field is still turbulent and exploratory, the crowd of researchers and industry professionals steering the direction of progress is converging upon a singular prevailing necessity—more compute.

Advancements to this end have seen great success, especially through the development of new dedicated hardware systems such as tensor processing units. However, new methods continue to demand ever-growing energy costs, computational resources, and training times. This results in an increasingly narrow application space, with bleeding-edge models such as OpenAI’s GPT-4 requiring days or even months to train and being commercializable only as cloud services to avoid inordinately expensive barriers to entry for consumers. As such, these approaches are impractical for more specialized use cases and for those where size and power consumption are of critical importance.

Embedded systems are a prime example of the types of applications that highlight these limitations. Consider design spaces such as wearable devices or autonomous

robots. The practicality of such technologies is dependent on their portability and form factor. Furthermore, their scope would be greatly limited by requiring uninterrupted access to the power grid or a high-bandwidth satellite signal. Thus, utilizing conventional general-purpose models with intense computational demands is often infeasible—a different approach is necessary.

1.1 Neuromorphic Computing

Neuromorphic computing is an alternative approach to computing that mimics that of the natural world: the brain. For traditional models, “neural network” is something of a misnomer—although represented as complex spatial graphs, they perform computation via a series of atemporal linear algebra operations, with each layer of nodes and edges representing parameters in those formulas. By contrast, neuromorphic models, known as spiking neural networks (SNNs), utilize asynchronous, event-based, decentralized processing where nodes (“neurons”) maintain individual internal states and communicate via discrete, temporal spikes across their connecting edges (“synapses”). These models often employ organic architectures rather than orderly layers, giving them a “hairball” appearance when visualized as a graph as in Fig. 1.1, not unlike that of an actual brain (1).

These attributes result in SNNs exhibiting a few key characteristics. First, by using organic, often recursive architectures, temporal processing, and neurons with persistent states, they are inherently capable of accumulating information in “memory”, which is useful in applications involving processing information over time. Second, their decentralized nature supports redundancy, a favorable feature in real-world applications where data often features noise and inconsistency beyond what preprocessing techniques can eliminate. Third, much like the human brain, they can achieve comparable results to conventional models with far fewer FLOPs/sec and are thus exceedingly energy efficient, with hardware implementations often boasting requirements on the order of milli- or microwatts (2; 3). This means they

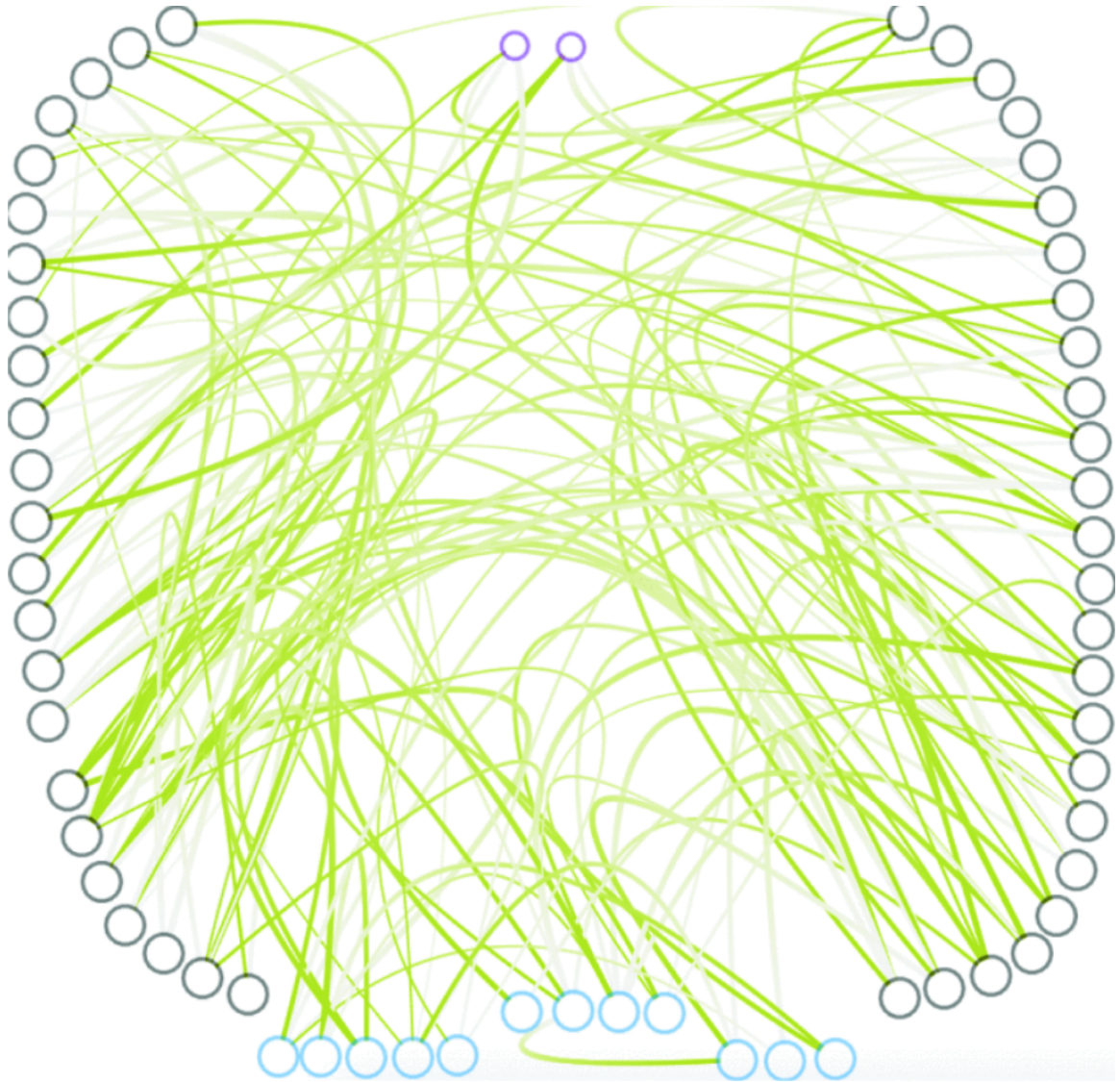


Figure 1.1: Example of a small SNN with an organic architecture. Input neurons are gray, output neurons are blue (bottom), and hidden neurons are purple (top). Synapse direction is indicated by shading, with spikes traveling from light to dark. Note that input and output neurons can both emit and receive spikes, allowing them to participate in computing rather than only encoding and decoding.

demand neither power-hungry processing units nor bulky batteries, lending them a key advantage on embedded hardware systems. However, they also do not scale well and fail to compete with conventional approaches on sufficiently complex problems that demand models with millions of neurons. Thus, the budding subfield has found its primary niche in embedded real-world applications, which tend to involve noisy time series data, and where slight performance dips are superseded by concerns of size and power demands.

1.2 The Simulation-to-Reality Gap

The key challenge that differentiates real-world applications from the rest of machine learning problems is the incomputable complexity of the real world. Agents must be trained over the course of thousands or millions of runs, if not more, to achieve adequate performance, so they are trained in simplified simulated environments where such large quantities of evaluations can be performed in a reasonable amount of time. This means that an agent is only as good as the simulator it is trained on. If the simulator is sufficiently inaccurate, no amount of training will produce a network capable of performing well in the real environment. The problem it is trained on and the problem it is intended to solve must be sufficiently similar for the network to achieve acceptable results. However, increasing the complexity of the simulated problem increases the length of the training process and the necessary complexity of the model. A balance must be found between accuracy and simplicity, where real-world deviations are small enough for the trained model to correct for when deployed.

Furthermore, the real world is not static. In many cases, the relevant conditions of the environment in which the agent is operating can change. Such applications demand that an agent perform well not in one problem, but rather a myriad of possible problems that are similar but distinct. As a simple example, consider the problem of implementing a cruise control mechanism for a boat, where an agent would keep the craft at a constant heading and speed. Surface currents, choppy waves, wind,

and a variety of other potentially relevant factors would alter how the state of the system evolves in response to any given action. A model trained with a simulation of still waters and calm winds would fail in these new conditions, no matter how accurate the simulation is.

The most obvious solution is to evaluate networks on many different simulated scenarios during each training epoch, thereby promoting generalization. But achieving generalizability requires exploding the complexity of neural architectures, exacerbating training times as well as the computational and energy demands of the model, while often sacrificing overall performance on any one scenario versus a specialized model trained for that specific set of conditions. Succinctly, a generalized model tends to be an expensive jack of all trades and master of none. While algorithmic and hardware advancements have partially mitigated the slowdown of complexity by utilizing clever linear algebra techniques and tensor processing units, they do not address the underlying fundamental problem. In addition, these palliative developments do not carry over to neuromorphic models.

1.3 Our Solution

In this work, we present an alternative approach to addressing the challenges of real-world applications in dynamic environments, utilizing the strengths of neuromorphic computing to train and evaluate an array of specialized expert agents and deploy them interchangeably as conditions change in real time. Due to their small size, SNNs can be trained and evaluated rapidly, allowing us to generate new networks for specific environmental conditions as they occur, with mere minutes of latency, or pretrain and store a large volume of networks for a breadth of possible conditions. We describe the development and implementation of NODES (Neuromorphic Optimization using Dynamic Evolutionary Systems), our software framework for facilitating all aspects of this process, and we demonstrate both the capabilities and limitations of this approach in its current state via two case studies.

Our goal with this project is to evaluate the viability of such an approach to dynamic real-world applications, with the intent of demonstrating that it allows small, lightweight neuromorphic models to handle a wide range of dynamic problems that they would otherwise be unfit for. We anticipated that the primary deciding factors for whether the NODES framework could handle a problem would be the rate of environmental changes, which could outpace our ability to track them, and the magnitude of inherent chaos of the system, which could make it too sensitive to slight deviations in those changes for our system to adequately approximate its behavior. Through our case studies, we were able to test these hypotheses. In addition, we discovered ways to address some of these issues and other considerations salient to the framework’s success.

The content of the remaining chapters is as follows. Chapter 2 is a brief survey of relevant work in the field pertinent to this exploration. Chapter 3 provides an overview of existing models, algorithms, and software utilized in this work, as well as the implementation of the NODES framework itself, with a focus on novel algorithmic aspects. Chapters 4 and 5 detail the case studies used to test and evaluate NODES. Note that though it is listed first, the case study in Chapter 4 was performed second to provide an example with less complexity than that of the following chapter. Chapter 6 discusses and compares the results of the case studies and their implications, highlighting key takeaways and revisiting hypotheses, as well as potential avenues for further exploration.

Chapter 2

Related Work

Evolutionary algorithms as an approach to training spiking neural networks is not a new concept. It was experimented with as early as 2005, as described by Pavlidis et. al. in (4), who used a parallelized version of the Differential Evolution algorithm developed by Storn and Price a decade prior (5). The algorithm was used to optimize the weights of SNNs for problems such as diagnosing diabetes from various medical readings or the simple Exclusive-OR problem. However, no modifications were made to the architecture of the networks—only scalar parameters were adjusted. A similar study was conducted the following year by Belatreche et. al. (6). As further investigations have been made in this direction, numerous architectural algorithms have been explored. In (7), López-Vázquez et. al. explore an approach to evolving non-recursive three-layer SNNs with an arbitrarily large hidden layer by representing each network as a word in a defined context-free grammar. In practice, trained networks did not exceed ten hidden neurons on any of the twelve benchmark datasets used. This approach is notable for being drastically different from the EONS algorithm, elucidating the potential for many yet-unexplored alternative evolutionary methods for SNN training.

The simulation-to-reality gap, or simply "reality gap", is a well-known and well-explored issue not only in machine learning but also in other fields dependent on

simulators, such as robotics. Much work has been done in developing methods of detecting and quantifying the reality gap. For example, in (8), Daza et. al. develop and test a framework for measuring the gap for model predictive control systems in the context of autonomous driving. Collins et. al. explore the more general case of evaluating which aspects of physics engines tend to have the greatest impact on the reality gap in (9), finding that the effects of different tuning parameters can vary between specialized simulator environments, although lateral friction, timestep granularity, and bounded actuator velocities appeared to be the most frequent culprits. In (10), Ligo and Birattari even demonstrate that common phenomena observed when transferring control software from simulation to reality also occur when moving between two different models. Within the realm of neuromorphic computing, research has been conducted on the possibility of closing the reality gap via online learning. In (11), Qiu et. al. demonstrate that synaptic plasticity can allow a network to adjust its weights and adapt to changes in real time. Their spiking models use a three-layer structure with recursive inter-layer synapses. The hidden layer is optimized with NEAT, a popular evolutionary algorithm for optimizing network weights and topologies. They also use NEAT to adjust the coefficients of a Hebbian plasticity rule on a per-synapse basis, optimizing its online adaptability. From the aforementioned works and many others, it is clear that addressing the reality gap is a pressing matter for the development of resilient autonomous control systems, neuromorphic or otherwise.

Chapter 3

Methods

In this chapter, we provide a technical description of the NODES framework following a higher-level overview of the foundational technologies employed and their germane advantages. This includes spiking neural networks, the EONS training algorithm used to generate them, and the TENNLab Neuromorphic Framework, a suite of tools, applications, and simulators that enables the accessible utilization and exploration of various neuromorphic methods. These technologies are discussed first to provide background information pertaining to the components comprising our system.

3.1 Spiking Neural Networks

As mentioned in Chapter 1, spiking neural networks (SNNs) are the primary type of model used in neuromorphic machine learning, in which nodes with intrinsic states transmit information via event-based spikes. This reflects the behavior of biological neurons and synapses. In a real brain, neurons accumulate potential energy ("membrane potential") until a threshold is reached, upon which they discharge and fire signals ("spikes") over connected synapses to other neurons, which each increase or decrease their potential accordingly. In this way, storage and computing are distributed across the entire collection of neurons and synapses. Note that synapse length affects the time required for a spike to travel between two neurons, hence the

use of temporal, event-based models. More complex neuron models exist, such as leaky neurons, which lose charge over time. However, in this work, we use simple neurons with integer states and thresholds that only change state when receiving a spike, and simple synapses with integer weights and delays that always carry spikes of the same magnitude.

Encoding and decoding spikes can be done with a variety of methods. Encoding converts an array of observation values into spikes, and decoding converts spikes into an array of action values. Some common decoding methods measure the rate of spikes or the time since the most recent spike from the output neuron. There are many different encoding methods. Some methods map each observation to a single input neuron, encoding a value as the magnitude or timing of a single spike or as a variable number of identical spikes at even intervals (a "spike train"). Others map each observation to multiple input neurons, with the observed value determining which neuron or neurons receive spikes.

SNNs can also feature a variety of architectures. They can be arranged in fully connected non-recursive layers in the same way as a conventional neural network, which is primarily used when converting such a network to a spiking model. However, neuromorphic applications often take advantage of the flexibility of SNNs, allowing neurons to be connected in any arbitrary manner as long as there is some path for spikes to propagate from input to output. The recursion and compactness this enables are beneficial traits for applications involving time series data and limited hardware resources, such as NODES. As such, the latter approach is used in this work.

3.2 EONS

Conventional neural networks are trained using the backpropagation algorithm, which involves computing partial derivatives of outputs and errors. This does not work for spiking neural networks due to the absence of a differentiable activation function and the presence of recursion. Thus, a different approach is needed.

Evolutionary Optimization for Neuromorphic Systems (EONS) is a framework developed at TENNLab to train SNNs (12; 13). It utilizes evolutionary algorithms, another subject in machine learning inspired by biology, in which a population of agents is repeatedly evaluated, culled based on performance, and repopulated with randomly modified versions of the surviving agents. Like the backpropagation algorithm, this process can continue indefinitely, but is typically limited to a set number of epochs. Unlike the backpropagation algorithm, EONS changes not only the parameters of the network, but the architecture itself—the random modifications, or mutations, include adding and removing nodes and edges. Therefore, this algorithm acts as both a parameter optimizer and a neural architecture search, allowing it to adjust the complexity of networks as necessary for a given problem.

In EONS, the starting population of networks is either randomly generated or seeded from a provided pool of networks. The networks are evaluated by running them on the simulator for several episodes and assigning them a fitness score using a fitness function that rewards the desired behavior. Then, a fixed percentage of the population is removed, with bias towards preserving networks with a higher score. Importantly, networks that perform poorly are not guaranteed to be eliminated—randomness is important for avoiding stagnation and continuing to explore the parameter and architecture space. The surviving networks are then copied and mutated to refill the population, and the process repeats.

For brevity, a full technical detailing of the EONS framework is not included here. An exhaustive description of its operations can be found in "Evolutionary Optimization for Neuromorphic Systems" and "An Evolutionary Optimization Framework for Neural Networks and Neuromorphic Architectures" by Schuman et. al., which present the framework’s initial conception and subsequent iterations (12; 13).

For a given application, we need to train networks on a variety of similar problems that differ only by varying the simulation parameters responsible for approximating changing real-world conditions. EONS is an ideal training approach for this, as it provides an evolutionary equivalent to transfer learning. Seeding the initial

population with our existing collection of networks, which have been trained on very similar problems, allows the algorithm to start with a population of networks that are significantly better at solving the desired problem than a randomly generated population. This considerably reduces the time required to train, allowing us to adapt to changing conditions much more rapidly.

3.3 TENNLab Neuromorphic Framework

TENNLab is a research group at the University of Tennessee, Knoxville that specializes in studying spiking neural networks, advancing the field of neuromorphic computing on both the hardware and software fronts. To this end, the group has developed the TENNLab Neuromorphic Framework—a suite of tools, applications, and simulators to enable accessible exploration of neuromorphic technologies and straightforward integration with external systems. The core components are built in C++, with interoperable wrappers and additional tools provided in Python for ease of experimentation. The Framework uses EONS as its primary method for training SNNs, making it a natural foundation upon which to build our work.

Two key aspects of the TENNLab Neuromorphic Framework are utilized in this work. First, we use its implementation of simulated SNNs, which support many neuron and synapse attributes for mirroring common neuroprocessor specifications, as well as multiple encoding and decoding schemes. Support for reading and writing these networks in JSON format is also provided. Second, the Polebalance application and associated control application driver enable experimentation with the training process and evaluation of SNNs on a simple simulated environment. This is one of our two case studies as described in Chapter 4. We were able to utilize these tools with minimal modifications to build the NODES framework and examine its capabilities.

3.4 NODES

Neuromorphic Optimization using Dynamic Evolutionary Systems (NODES) is the core software package that enables real-time learning and adaptation. It is designed to be a general-purpose, highly configurable, comprehensive system that handles the training, evaluation, storage, and deployment of networks, as well as the continual optimization of simulator parameters, while facilitating ease of integration with any neuroprocessor and application. The various functions it fulfills are separated into standalone processes that communicate via local sockets to coordinate their operation, mostly through a central “Coordinator” process. Furthermore, it is designed to be sufficiently lightweight to run on small, portable processors, while also being capable of scaling the training functionality to use the extra computing power provided by larger systems with many cores. The end user only needs to provide a parameterized simulator, a fitness function, a communication protocol for their chosen neuroprocessor, and their desired specifications for the various EONS and NODES settings. Fig. 3.1 depicts a simplified diagram of the interconnected processes that comprise NODES. These processes each independently handle one or multiple core functions of the framework, allowing them to act as black boxes to the other components.

3.4.1 Training

The EONS Interface and Sim Scheduler components designed by Karan Patel facilitate training networks using the EONS algorithm. The EONS Interface handles the core functions of the algorithm, except for evaluating the networks in simulation. This is delegated to the Sim Scheduler, which receives requests to run one network or a population of networks for an episode each on the simulator with a specified set of simulation parameters. Evaluation is performed in a separate process to allow for the networks to be loaded and run on a physical neuroprocessor—normally, EONS only supports simulated neuroprocessors. The resulting fitness scores from these

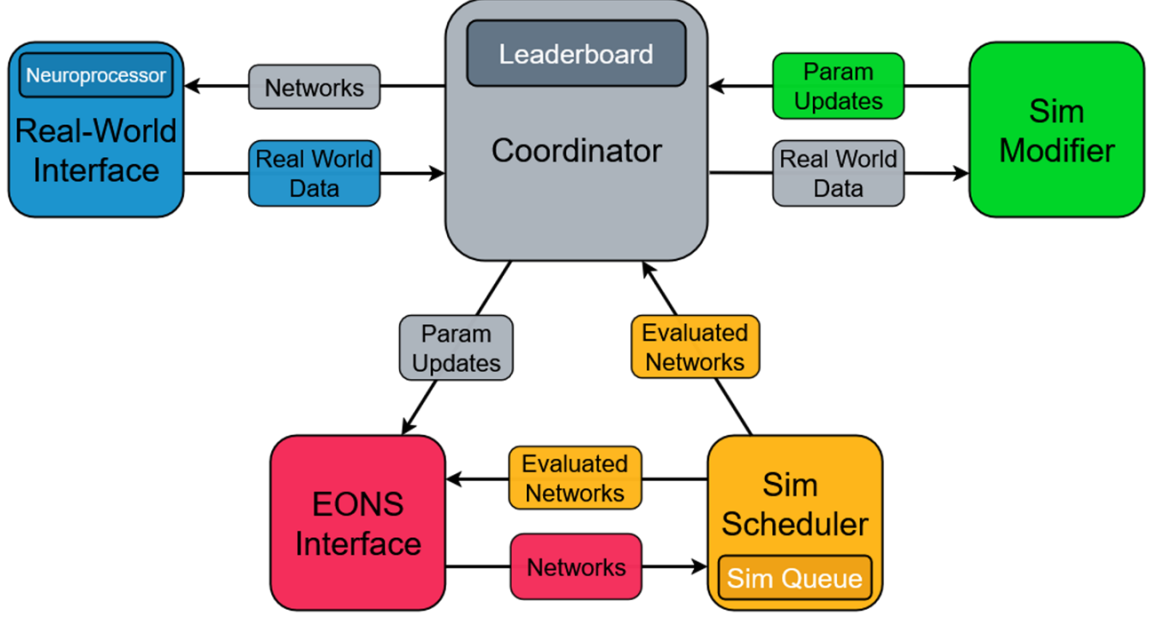


Figure 3.1: Simplified flowchart of NODES processes.

evaluations, along with the networks and parameters, are sent back to the EONS Interface to continue the training process. At the end of a training round, the highest-scoring network is sent to the Sim Scheduler for evaluation. That network and its results are then sent to the Coordinator and stored for later queries. The EONS Interface also periodically receives new simulation parameter values from the Coordinator, which are passed along to the Sim Scheduler in subsequent training rounds. In this way, these two components enable a continuous flow of fresh networks trained on any provided simulation parameters to the Coordinator, allowing NODES to adapt to changing real-world conditions by deploying specialized networks for those specific conditions.

The Sim Scheduler also provides the ability to scale the number of processes dedicated to training to the available hardware resources. Each request received by the main Sim Scheduler process is sent to a separate “Worker” process, which handles running an instance of the simulator, allowing multiple evaluations to be performed concurrently as shown in Fig. 3.2. The maximum number of these processes is

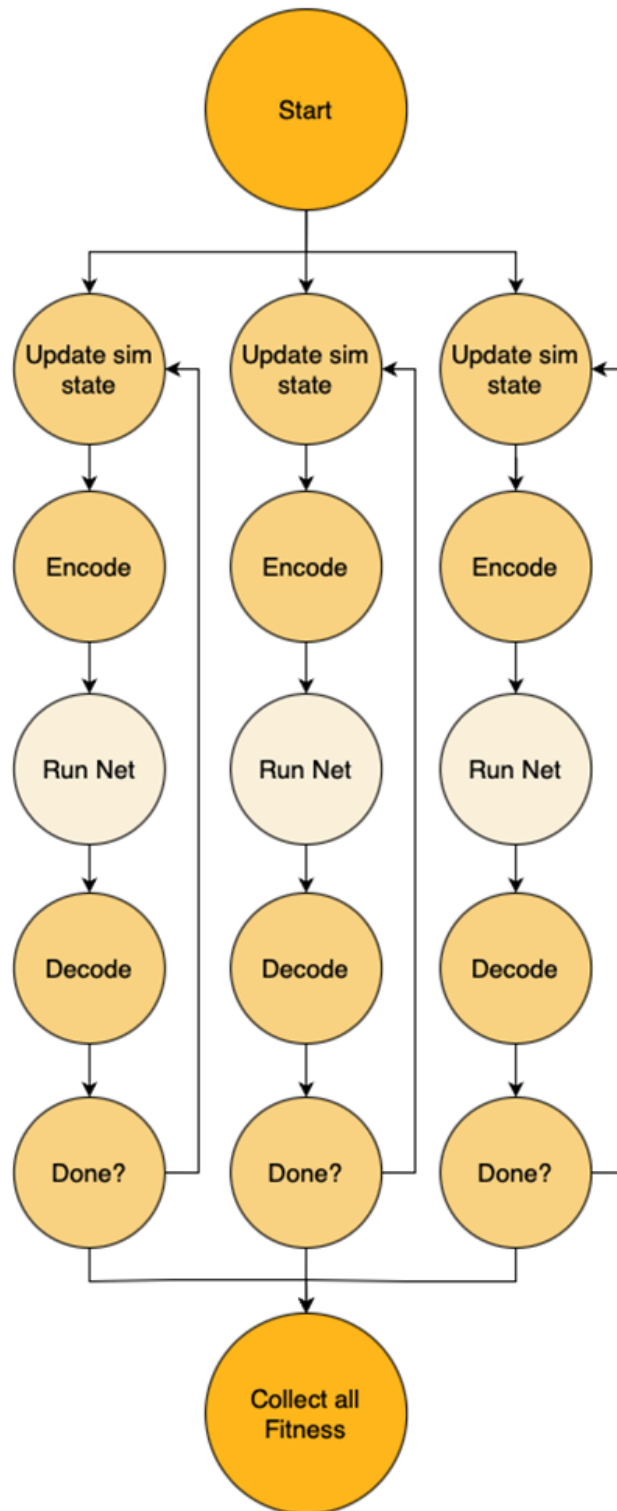


Figure 3.2: Sim Scheduler runs multiple simulators in parallel.

specified in the NODES settings. The results of the completed episodes are then sent to a “Collector” process, which passes them on to the EONS Interface or Coordinator.

3.4.2 Parameter Optimization

As mentioned above, the Sim Scheduler runs the simulator with specific parameters to train networks that are specialized to perform well under specific real-world conditions. However, this requires that the chosen simulator parameters be analogous to those conditions. Thus, for any real-world conditions, NODES must be able to determine a suitable set of parameter values for which the simulator adequately approximates the behavior of the real-world system so that the training procedure generates networks with the desired behavior.

This is achieved using the Sim Modifier process as implemented by Tyler Nitzsche. The Coordinator periodically sends batches of observations and actions collected from the neuroprocessor controlling the physical system to the Sim Modifier. This process then employs Bayesian optimization via the Optuna hyperparameter optimization framework to determine parameter values for which those actions result in similar observations on the simulator. First, an instance of the application simulator is initialized to a specific state corresponding to the first observation in the batch. Then, it is run with the actions from the batch as input. The resulting observations are compared to the recorded observations to calculate a mean squared error loss metric quantifying the difference between them. This process, visualized in Fig. 3.3, is repeated numerous times with different simulation parameter values determined by the Bayesian optimization algorithm as it gradually converges towards an unknown optimum.

Parameter values that result in a low loss correspond to a simulator configuration that closely mimics the behavior of the real-world system. These parameters thus serve to approximate the conditions of the real environment, and networks that perform well in simulations using those parameters can be expected to perform

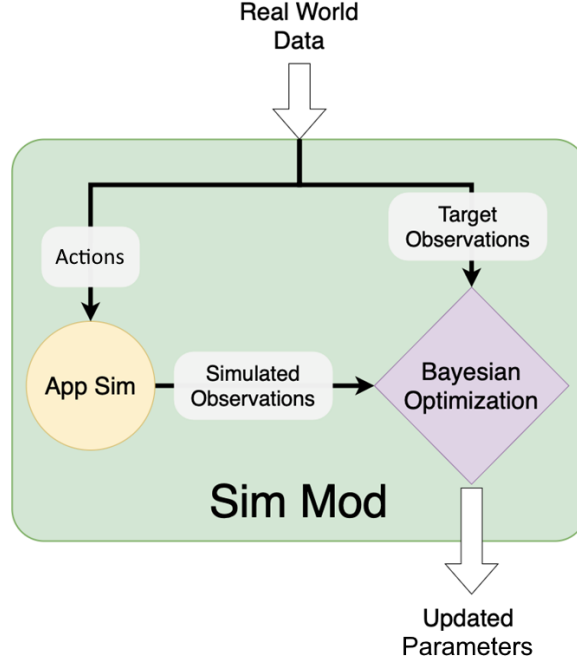


Figure 3.3: Flowchart of how the Sim Modifier process approximates matching simulation parameters from real-world data.

well when deployed on the physical system. After several runs, the Sim Modifier sends the parameters that achieve the lowest loss to the Coordinator, where they are recorded and passed on to the EONS Interface to ensure that networks are trained on a simulator analogous to the current real-world environment.

3.4.3 Interfacing with Hardware

The NODES framework needs to be able to communicate between the hardware it is hosted on and the hardware controlling the real system, i.e., the neuromorphic neuroprocessor chip. The Real-World Interface, designed with the help of Karan Patel, encapsulates this functionality. It handles loading SNNs received from the Coordinator onto the neuroprocessor, as well as collecting batches of real-world data from the neuroprocessor to send to the Coordinator for parameter optimization and future network selection.

NODES is intended to be flexible and capable of being used in as wide a variety of applications as possible, so we do not implement a specific communication protocol. Rather, we provide a template interface for implementing communications between NODES and any arbitrary neuromorphic hardware component on a per-application basis. Furthermore, as running the NODES framework separately from the real system is often necessary for tasks such as pretraining networks, hyperparameter optimization, testing, and debugging, the Real-World Interface also supports the use of a simulated environment as a proxy for a physical system.

As the Real-World Interface receives per-timestep data from the neuroprocessor, it collects observations, actions, and cost values and sends them in periodic batches to the Coordinator. This enables parameter optimization in the Sim Modifier as well as continual performance evaluation of the deployed network by the Coordinator, which is responsible for deciding when to replace it with a new network.

3.4.4 The Coordinator

The Coordinator process is the central organizing component of the NODES framework, responsible for multiple core functions that allow the system to function cohesively. It enables communication between the other arms of NODES, as well as reading and writing trained networks and their recorded scores to and from storage. Crucially, the Coordinator is also responsible for populating and querying the Leaderboard data structure, which decides when to replace the network currently deployed on the real system and which network to replace it with. The remainder of this section details the implementation of this data structure and its justifications.

Whenever a trained network is received from the Sim Scheduler, the Coordinator stores it in an associative array keyed on an MD5 hash of the network itself, which is treated as a unique ID for that network. The ID is then stored in the Leaderboard, along with its evaluated fitness score and the simulator parameters on which it was trained and scored. The Leaderboard can then be searched for high-scoring

networks within a range of acceptable training parameters. When the Coordinator receives a score for the currently deployed network from the Real-World Interface, it searches a region of the Leaderboard with training parameters similar to the current approximated real-world parameters and fetches the network with the highest score within that region. If its score is higher than the real-world score, and it is not the same network that is currently deployed, it is sent to the Real-World Interface to replace the deployed network. However, we anticipate that networks will tend to perform slightly worse in the real world than in simulation, as neither the simulation nor its approximated parameters will be perfect, so a scalar bias is applied to the real-world score before comparison. This reduces the likelihood that the deployed network will be replaced by one that performs worse.

The most basic implementation of the Leaderboard would be to store each network’s ID and score at N-dimensional coordinates in a continuous parameter space corresponding to the parameter values on which that network was trained, where N is the number of simulator parameters available for the given application. When querying this Leaderboard, a region of any arbitrary shape could be searched, such as a circle as shown in Fig. 3.4. However, this approach becomes increasingly inefficient as the number of stored scores increases, as every entry in the Leaderboard must be checked to determine whether it is within the search region.

Instead, we implement the Leaderboard using a discrete approach. For each parameter, the range between its minimum and maximum allowed values is evenly divided into several sections. The number of sections is defined for each parameter on a per-application basis. The parameter space is thus divided into a grid of distinct regions, or "bins". Rather than storing each ID and score at a unique key, they are stored in a list of all entries that fall within the same bin in parameter space, sorted by score. Queries are performed by checking a collection of bins rather than a region, and only the highest-scoring network in each bin must be checked. This approach is shown in Fig. 3.5. As long as each parameter has enough bins to achieve a sufficient level

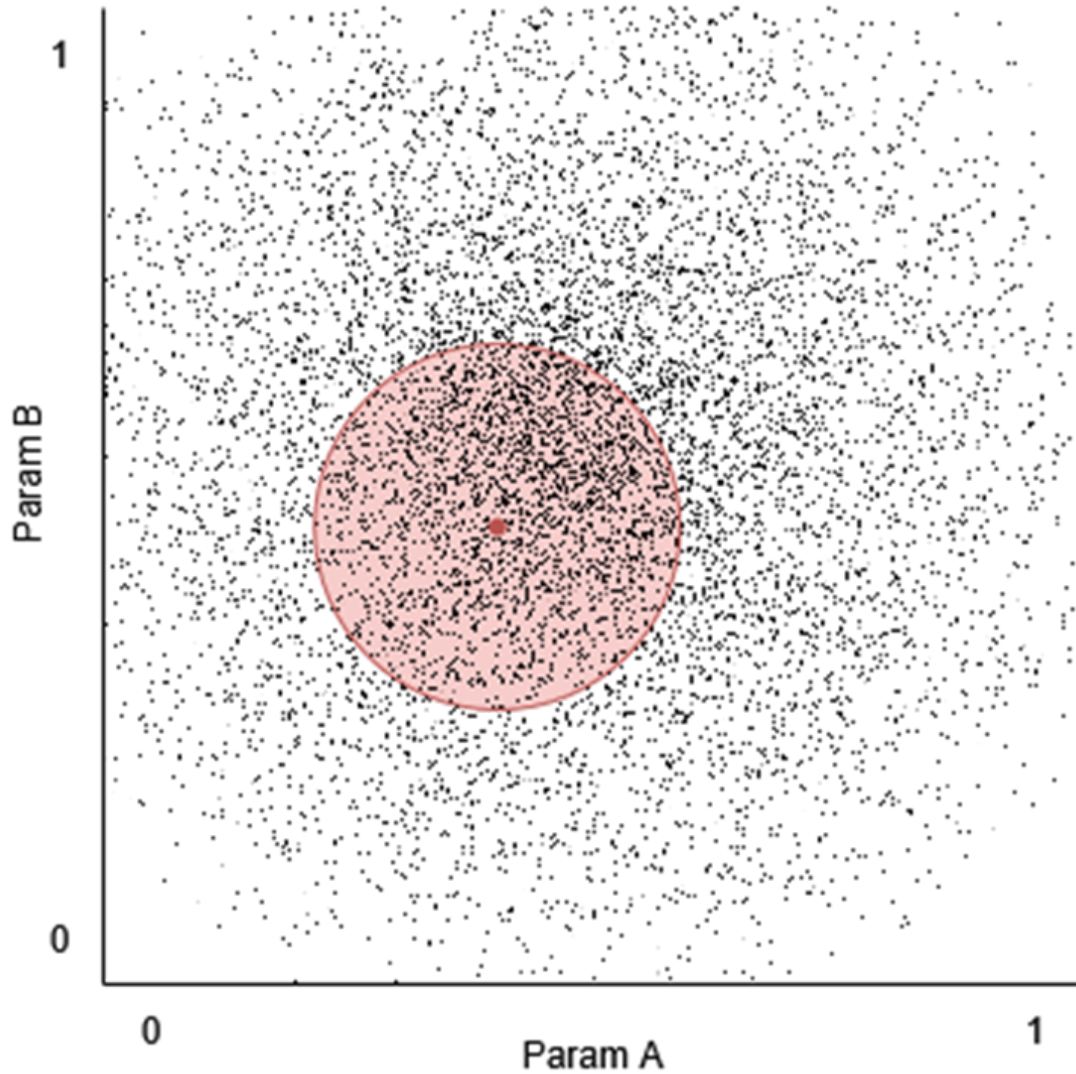


Figure 3.4: The naive method of implementing the Leaderboard. Each dot represents an evaluation of a network, with the position of the dot indicating the simulation parameters used for evaluation. The red circle visualizes a radial search range of parameter values around the approximated real-world values, represented by the red dot.

of granularity, this implementation of the Leaderboard functions nearly identically to the continuous version while significantly reducing the computational complexity.

The Leaderboard can also be seeded on startup with networks that have been trained beforehand. This helps improve responsiveness to sudden changes in real-world conditions by eliminating the need to wait for a new network to be trained for the new conditions. Instead, a pretrained network can be deployed as soon as the Sim Modifier detects the change and generates a parameter approximation. The EONS Interface will continue to train additional networks in the background.

Lastly, following the Engine Control case study, we developed a final improvement to the Leaderboard—a second Leaderboard. In addition to storing the score each network achieved on the simulator, the Coordinator also stores the scores for deployed networks received from the Real-World Interface. These are stored in a separate “Real-World Leaderboard”. Networks deployed for an extended duration may receive multiple scores across different sets of parameters. Therefore, the Real-World Leaderboard can contain an entry for the same network in each of several bins, with each entry being treated independently. However, each bin contains only a single entry for each network, consisting of the network’s ID and the average of all scores inserted into the bin for that ID. In this way, NODES can remember when a network performs well on a set of real-world conditions and redeploy that network the next time those conditions arise. Furthermore, it can avoid redeploying networks that scored high in simulation but performed poorly on the real system. Over time, the Real-World Leaderboard will continue to fill up with data, allowing the system to immediately swap to a high-quality network whenever the Sim Modifier detects conditions that have previously been encountered. Like training scores, real-world scores are written to file as metadata alongside the network itself so that even if NODES is stopped, the collected data will be retained for use the next time the framework is used.

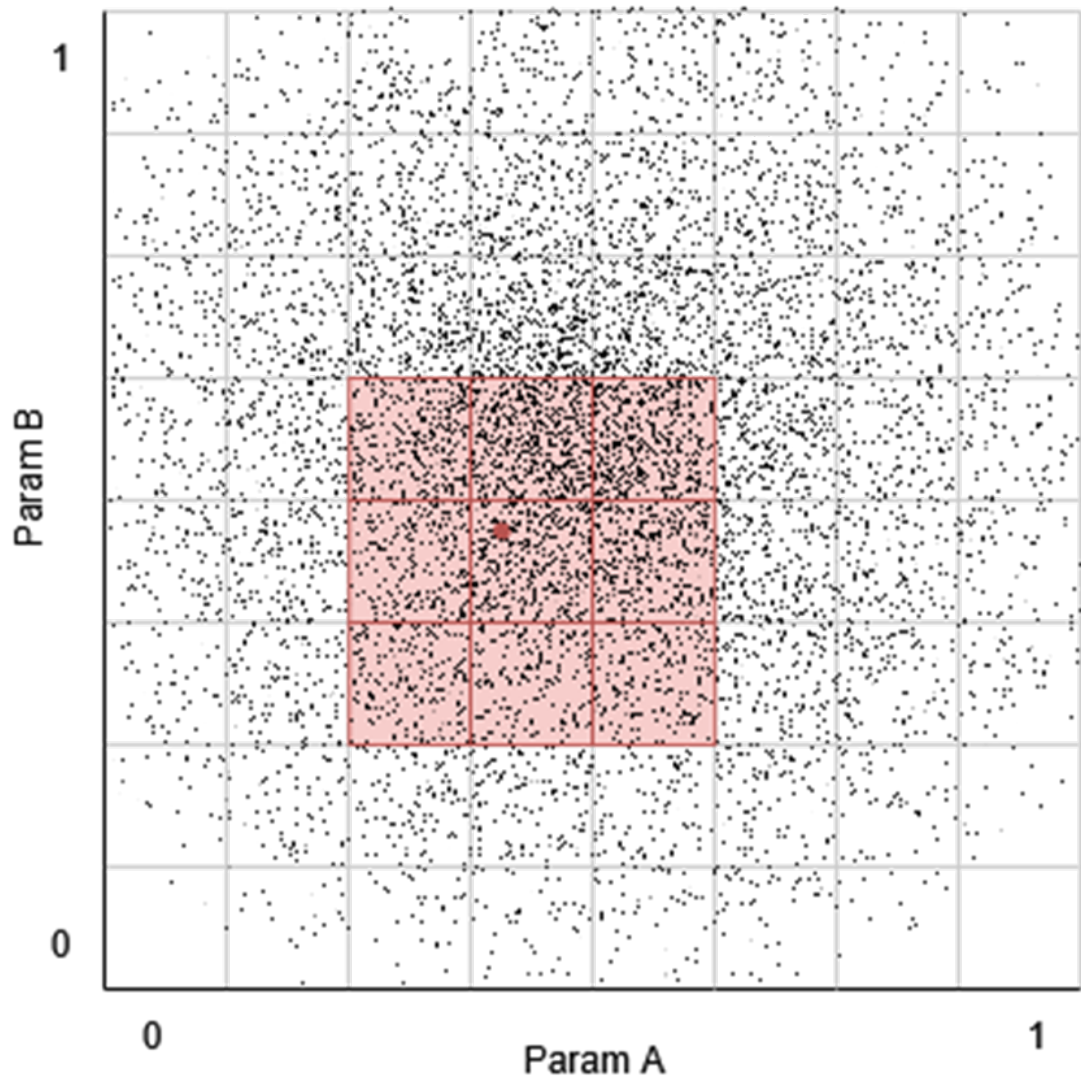


Figure 3.5: The binned approach to the Leaderboard. Gray lines delineate each bin. Note that although the search region contains many entries, only one needs to be checked in each bin.

Chapter 4

Case Study 1: Polebalance

To examine the effectiveness of the NODES framework and whether it behaves as expected in a controlled environment, we use the Polebalance application. This is a simple simulated application from the TENNLab Neuromorphic Framework intended as a benchmark for training algorithms and neuroprocessors (14). It is based on the cart-pole system first described by P. E. K. Donaldson in 1960, which has since been replicated and used in many studies throughout the field of machine learning as a canonical problem for testing learning models (15). This application was chosen for its simplicity, which allows for short training times and small network sizes, as well as its transparent and deterministic nature, eliminating randomness or unseen variables as a source of error when testing. Networks were trained and tested using a simulated RISP processor as described by Plank et. al. in "The RISP Neuroprocessor – Open Source Support for Embedded Neuromorphic Computing" (16).

In the Polebalance application, the goal is to balance a pole affixed atop a cart by moving the cart. The base of the pole is fixed in place in the center of the cart, but the pole is free to rotate clockwise and counterclockwise. In our simulation, a constant horizontal force is applied to the cart, either to the left or right, as decided by the agent controlling the system. This is the only manner in which the agent can affect the system, so the action space consists of a single binary output. The

observation space consists of four values—the position and velocity of the cart, and the angle and angular velocity of the pole.

The remainder of this chapter describes the configuration and modification of the Polebalance application for testing the adaptability of the NODES framework, followed by an examination of its performance and behavior when running for an extended duration in a changing environment.

4.1 Simulator Details

TENNNLab’s implementation of the Polebalance application is episodic—it runs until the agent fails or a maximum number of timesteps elapse. An episode will terminate early if one of two failure conditions is met. The first condition is met when the angle of the pole’s rotation from vertical exceeds a maximum value. This threshold is typically configured to be an angle beyond which the system is unrecoverable. The second condition is met when the cart moves outside of a legal region, which represents the bounds of the length of the track upon which the cart rolls. If the cart were allowed to move infinitely in either direction, the problem would be much easier, as the agent would always be able to push the cart in whichever direction the pole is falling until it became upright. By restricting it to a finite space, the agent must balance the pole while keeping the cart away from the edges of the track.

Various parameters are available for adjusting the simulated environment. In addition to the bounds of the failure conditions, users can also modify physical parameters, such as the masses of the cart and pole, as well as the frictional and gravitational forces they experience. Furthermore, we modified the application to support an optional “wind” force on the cart and pole. Note that we did not implement a realistic fluid drag simulation, as this would slow down the simulator significantly and is not necessary for the demonstrative purposes of this case study. Instead, we represent wind as constant forces on the cart and pole.

We adjusted two parameters while running NODES—the direction and intensity of wind, and the friction of the track on the cart. These serve as a simple and intuitive representation of real environmental changes, such as wind and rain. Friction values ranged from 0 to 2, indicating the Newtons of force exerted on the cart opposite to its current direction of motion. Wind values ranged from -5 to 5, indicating the Newtons of force exerted on the cart at all times, with negative and positive values indicating the force pushing to the left or right, respectively. The wind force exerted on the pole is one-tenth as much, representing the smaller cross-sectional area of the pole. A 10-by-10 binning scheme was used for the Leaderboard, corresponding to intervals of 0.2 N and 1.0 N for friction and wind, respectively. All other constants were unchanged across all tests. Cart and pole masses were 1 kg and 0.1 kg, respectively, and the pole was allowed to rotate with no friction. Gravitational acceleration was 9.81 N. The external force exerted on the cart was 10 N, with the agent being queried for the direction of this force every 0.02 seconds. The maximum allowed angle of the pole was 12 degrees, and the width of the cart track was 4.8 meters.

By default, the Polebalance application has a maximum episode length of 15,000 timesteps, and the fitness function for evaluating performance is simply the duration of an episode—the longer a network can keep the cart-pole system alive, the higher the score it receives. To demonstrate that modifying the friction and wind parameters of the simulator within these ranges has a significant impact on its behavior, we trained and tested networks on a variety of parameter values. Nine pairs of friction and wind values were chosen. For each, networks were trained on those parameters using EONS until we had 20 networks that achieved average scores of at least 5,000 across 100 episodes. Each of these 180 networks was then tested for 100 episodes on 121 different pairs of friction and wind values. Fig. 4.1 shows the average scores of each of the nine batches of 20 networks on the tested parameter configurations. As expected, networks perform well when tested under conditions similar to those under which they were trained, but perform poorly under conditions that are dramatically different.

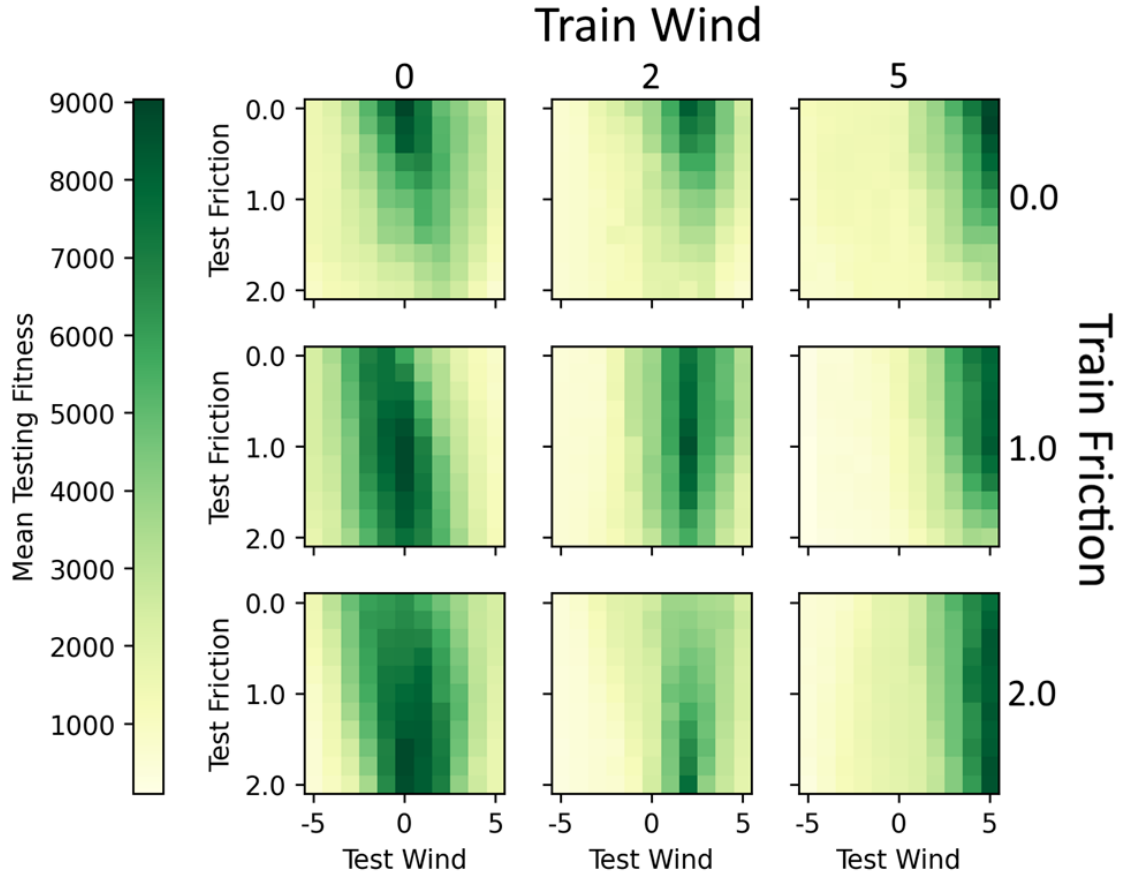


Figure 4.1: Mean scores of collections of high-scoring networks trained on Polebalance with specific params on a variety of different parameters. Note the trend of scores decreasing as the parameter values move further from the training values.

This bulk training and testing also allowed us to experiment with different encoder and decoder schemes. The encoder that performed best is a spike train encoder referred to in the TENNLab Neuromorphic Framework as "Fred". It is a two-bin encoding technique, meaning there are two neurons for each observation value. If an observation value is in the lower half of the expected range, the first neuron receives a spike train, and if it's in the upper half of the expected range, the second neuron receives the spike train. The number of spikes is proportional to the distance of the observed value from the center of its range, with a maximum of 8. For example, if the observed pole angle is 6 degrees to the right, the encoder will apply 4 spikes to the second pole angle input neuron. The decoding technique is much simpler—there are two output neurons, one for "left" and one for "right", and whichever receives more spikes is the direction the agent chooses to push the cart. The rest of the synapses and neurons in the network exist to convert the input spikes to output spikes in a way that steers the cart such that the pole remains upright and the cart does not slide off the track.

When testing NODES, we reduced the maximum episode duration from 15,000 to 1,000 timesteps to allow for more frequent evaluations of deployed networks. Anticipating that this change would reduce the difficulty of the problem, we also implemented a more punishing fitness function to ensure that trained networks would consistently perform well. Normally, the reward at the end of an episode is equal to the number of timesteps elapsed—an agent that survives for 1,000 timesteps would receive a score of 1,000 for that run. We added two penalty mechanisms to decrease the reward for each timestep proportionally to the angle of the pole and the position of the cart.

$$\text{reward} = \sum_{i=1}^{N_{steps}} \frac{|\theta_i|}{\theta_{max}} \cdot \frac{|x_i|}{x_{max}} \quad (4.1)$$

For example, let $\theta_{max} = 12$ and $x_{max} = 2.4$. If at timestep i , the pole is tilted 6 degrees to the right ($\theta_i = 6$), the reward for that timestep is halved from 1 to 0.5.

Then, if the cart is 1.2 meters left of the track’s center ($x_i = -1.2$), the reward is halved again to 0.25. Thus, to receive a good score, an agent must not only survive for 1,000 timesteps, but also keep the pole upright and the cart near the center of the track while doing so. When testing this modified Polebalance application separately from NODES, we found that trained agents were able to consistently survive for the full episode duration and achieve average scores of 600 or higher.

4.2 Results

This section examines the results from two recorded test runs—one with only a single pretrained startup network, and one with a batch of pretrained networks spanning the full range of allowed parameter values. The resilience and recovery time of the NODES framework are examined by manually changing the parameters of the simulator used in place of the ”real world” and recording the scores of the deployed network. In each test, the same set of parameter configurations is used. Note that the difference between each subsequent pair of parameters increases throughout the tests. This allows us to examine the impact of smaller versus larger changes in real-world conditions on the behavior of NODES. The Sim Modifier process consistently generated parameter approximations roughly every 7 seconds and rarely had to discard any due to high loss, resulting in minimal latency between the manual parameter changes and their detection by NODES. The EONS Interface and Sim Scheduler completed a training run every 90 to 100 seconds and consistently produced networks with simulation scores of at least 600, with the majority exceeding 700.

Fig. 4.2 shows the results of the first test. Note that the first trained network scores high enough in simulation to replace the startup network in the Real-World Interface. However, it nonetheless performs somewhat inconsistently. Eventually, a particularly low-scoring episode causes it to be replaced by a network trained on the new parameters. The new network performs well for long enough that the random low score 15 minutes later does not lower its average real-world score enough to

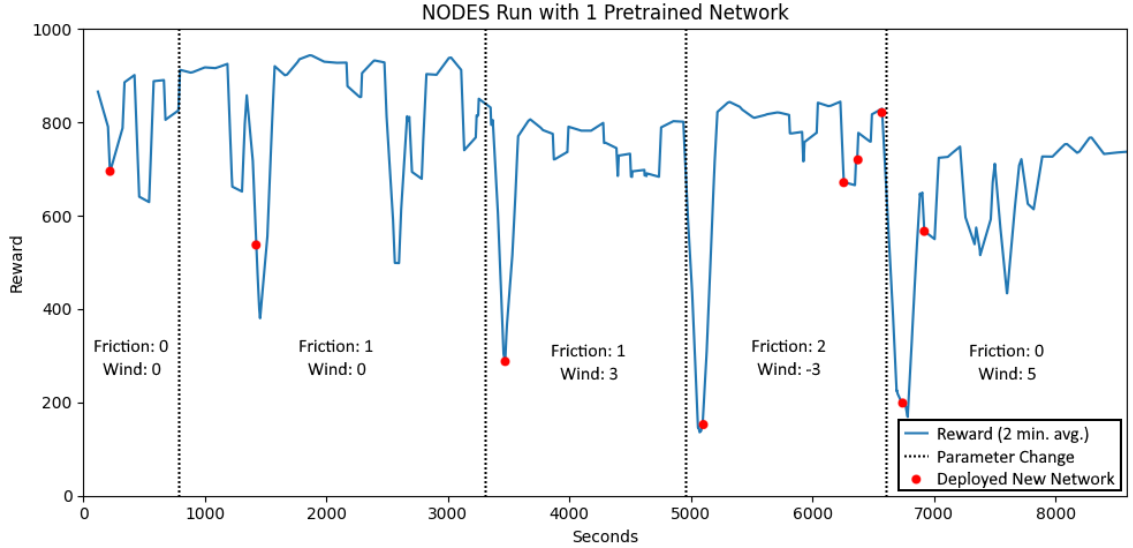


Figure 4.2: NODES is run for roughly 2 hours and 20 minutes on the Polebalance application. The Leaderboard is not populated with any pretrained networks except for the single necessary startup network. The blue line shows the average per-episode reward of the preceding two minutes over the duration of the test. Dotted vertical lines indicate when the parameters were manually changed, and red dots indicate when the deployed network was swapped.

trigger an immediate replacement. However, when wind is added at 3,300 seconds, its performance drops dramatically, and as soon as a network is trained on the new parameters, it is deployed. Following this, the average score quickly recovers and stabilizes. This behavior occurs again when the next parameter change occurs at 4,950 seconds. The final and largest parameter change at 6,600 seconds causes an extreme dip in performance that takes longer to recover from. From this data, we verify our hypothesis that NODES is able to easily adapt to small changes, but sudden, drastic changes in conditions will greatly reduce performance until networks can be trained on them.

The results of the second test are shown in Fig. 4.3. There is an immediately apparent difference in the performance of the NODES framework when pretrained networks are provided. Scores are slightly higher and much less inconsistent under all conditions. Furthermore, abrupt parameter changes do not disrupt performance

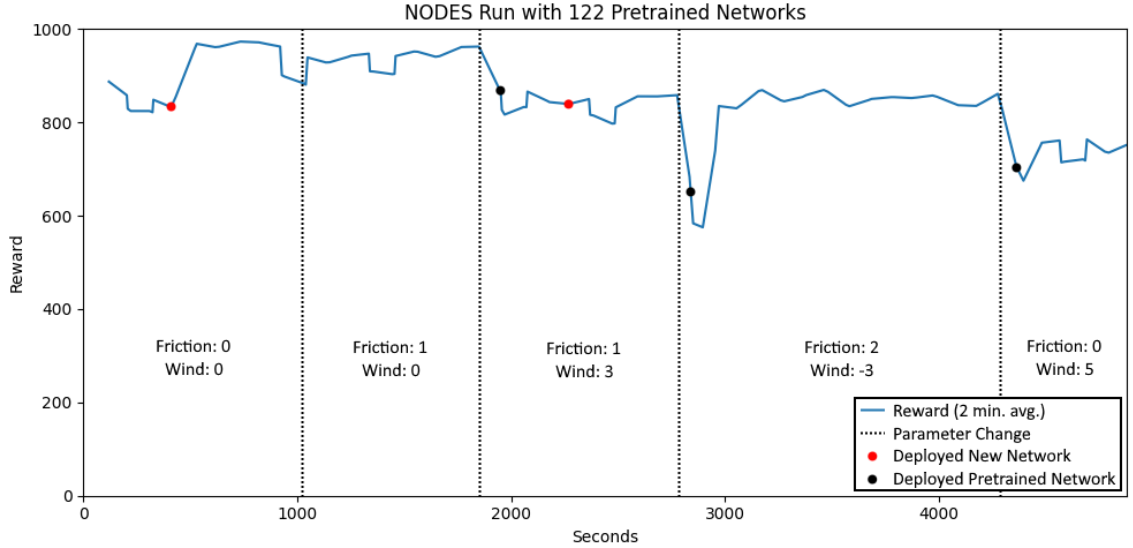


Figure 4.3: NODES is run for roughly 1 hour and 20 minutes on the Polebalance application. The Leaderboard is populated with 121 pretrained networks so that for all possible parameters configurations, it contains one or more networks that perform well. Black dots represent a network from the pretrained batch being deployed.

nearly as much. Note that although pretrained networks are immediately deployed after parameter changes, in some cases they are replaced by another newly-trained network.

This test demonstrates the immense advantage provided by a pretrained batch of networks. The batch used for this test took approximately 2 hours to train on a personal laptop, highlighting the benefit of a simple simulator with a small number of low-sensitivity parameters—training a batch with twice as many values per parameter would quadruple the time required, and adding a third parameter would increase it by a factor of ten. In contrast, the pretrained batch used for the next case study had to be trained overnight on a computing cluster at Oak Ridge National Lab.

Note that for both tests, average scores decrease on more extreme wind values. This is because the problem becomes inherently harder as wind increases, so networks trained on extreme winds tend to perform worse than those trained without wind even when tested on their training parameters. Adaptability will only get you so far.

Chapter 5

Case Study 2: Engine Control

This chapter includes content from "Evolution at the Edge: Real-Time Evolution for Neuromorphic Engine Control" by Patel et. al., a conference paper at NICE 2025 on which Ethan Maness is a co-author (17).

The second case study expands upon prior work at Oak Ridge National Laboratory (ORNL) for applying neuromorphic solutions to the problem of hybrid diesel-ammonia engine control (18). Much of this thesis work is a continuation of the ideas from that investigation. This study was conducted in collaboration with researchers at ORNL with funding from the UT-Oak Ridge Innovation Institute seed program. Alongside Karan Patel, Ethan Maness, and Tyler Nitzsche from the University of Tennessee, four collaborators from ORNL were involved in this project. Bryan Maldonado and Brian Kaul handled the operation and maintenance of the physical engine as well as the creation of the simple simulator used to approximate its behavior. Brett Witherspoon and Aaron Young developed the "FireBox" platform for facilitating communication between the Raspberry Pi-5 and BeagleV-Fire single-board computers, which host the NODES framework and μ Caspian neuroprocessor respectively, as well as communication between the neuroprocessor and the engine control unit (ECU).

In this case study, we endeavor to use NODES to autonomously control the diesel intake of a dual-fuel combustion engine which uses diesel fuel as a starter for burning gaseous ammonia. The engine used for this study is a Cummins ISB single-cylinder engine, as shown in Fig. 5.1. While the automobile industry is becoming increasingly electrified, these types of engines are used for large-scale transportation vehicles such as locomotives and cargo ships where electric power is still infeasible. Because of this, reducing the environmental impact of these engines will continue to be an important consideration in the near future. Our goal is to use NODES to reduce diesel usage while maintaining stable and efficient combustion to lower greenhouse emissions. This also has the effect of improving the cost effectiveness of the engine, making it marketable to companies using these engines.

However, this application is considerably more complex, chaotic, and opaque than the Polebalance application in the previous chapter. The internal state of the engine is a dynamic system which varies not only on a minute-to-minute basis, but also over a period of months as residual buildup accumulates and components experience wear. The combustion system is far too complex to directly model with our limited computing resources, so our simulator must be simplified to an exceptional degree. The system also constitutes a feedback loop—the internal conditions alter the impact

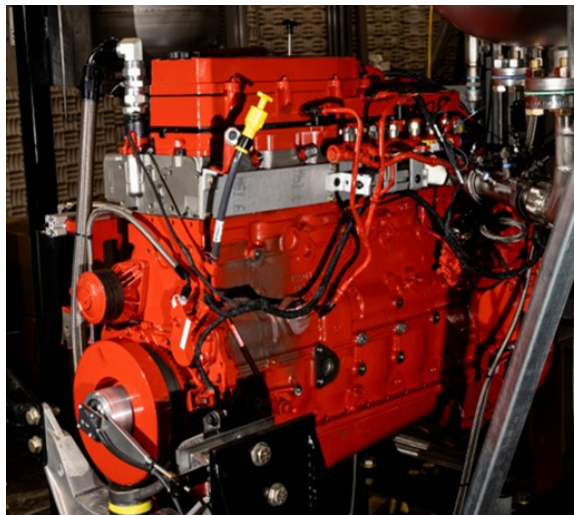


Figure 5.1: Cummins ISB 6.7L Engine

of the diesel intake variables, which then alter the internal conditions—making it less stable and predictable. Furthermore, the readings we receive from the ECU are not exact values, only approximations of the masses of each fluid within the engine.

5.1 Problem Specifics

For this application, we use three observations, those being the estimated masses of diesel, ammonia, and air within the engine once per cycle, with a cycle rate of roughly 600 rpm, giving a timestep duration of roughly one-tenth of a second. These values are normalized from their expected ranges to be between -1 and 1. The expected ranges are 6.3 to 9.4 mg of diesel, 67 to 76 mg of ammonia, and 1.6 to 1.7 g of air. The neuroprocessor outputs two action values each cycle, which are scalar values between -1 and 1 indicating adjustments to the default diesel injection mass and timing to make for that cycle. These defaults may be modified as seen fit by the engine operator, but for our purposes we assume that the diesel injection mass is 7.2 mg and the start of diesel injection occurs at a crank angle of 46 degrees before top dead center (bTDC) during each cycle. The neuroprocessor adjusts these values by ± 1 mg and ± 1 degree, respectively.

The cost function was iterated on for several days in testing to find a set of coefficients for which trained networks would achieve the desired performance. It is designed to balance three variables—diesel mass M_d , combustion efficiency η_c , and combustion timing CA_{50} as measured by the crank angle bTDC at which 50 percent of fuel has been burned, which is an indicator of the point when the majority of heat is released during combustion. The latter two variables affect the performance of the engine, while the first affects the emissions and cost efficiency. The final coefficients are $\alpha = 2.52/\text{mg}^2$, $\beta = 3.0$, and $\gamma = 0.133/\text{deg}^2$.

$$\text{cost} = \alpha M_d^2 + \beta(1 - \eta_c) + \gamma(CA_{50} - 3)^2 \quad (5.1)$$

5.2 Engine Modeling

The simulator consists of a series of matrix operations designed to emulate the internal state dynamics of the combustion engine. The detailing and explanation of these operations is beyond the scope of this work—refer to the aforementioned NICE 2025 conference paper for mechanical engineering details (17). It is sufficient to state that while these equations do not constitute a proper physics engine, they are informed by combustion engine theory. The model was calibrated using data collected on the same engine roughly a year prior to testing the NODES framework. In Fig. 5.2, we compare the collected data and the behavior of the simulator. Notice that the simulator follows the same general curve as the real data, but has much less chaotic mass values. This is due both to noise in the data collection process and simplification of the engine dynamics by the simulator.

The engine has diesel fuel direct injection and was fitted with gaseous ammonia port fuel injection. The ammonia (NH_3) port fuel injection quantity was fixed at a rate of 42 g/min, while the diesel injection quantity and timing were manipulated between 6–9.5 mg and 39–45 deg bTDC, respectively. A total of 42 operating conditions was experimentally recorded to build a robust combustion model. The heatmap in Fig. 5.3 shows the combustion variability (less is better) measured by the coefficient of variation (CoV) of the indicated mean effective pressure (IMEP). According to industry standards, a CoV of IMEP lower than 3% is desirable for avoiding unnecessary noise, vibration, and harshness. The black solid line shows the location of the optimal CA_{50} . Note that this optimal value occurs in regions of high CoV. Therefore, the controller needs to adjust the diesel injector parameters, such as quantity and timing, to find a balance between optimal combustion location and lower cycle-to-cycle variability.

Due to the conditions of the real engine being too complex and chaotic for our model to capture, we instead opted to use "black box" simulation parameters. The parameters only need to ensure that the simulator behaves similarly to the physical

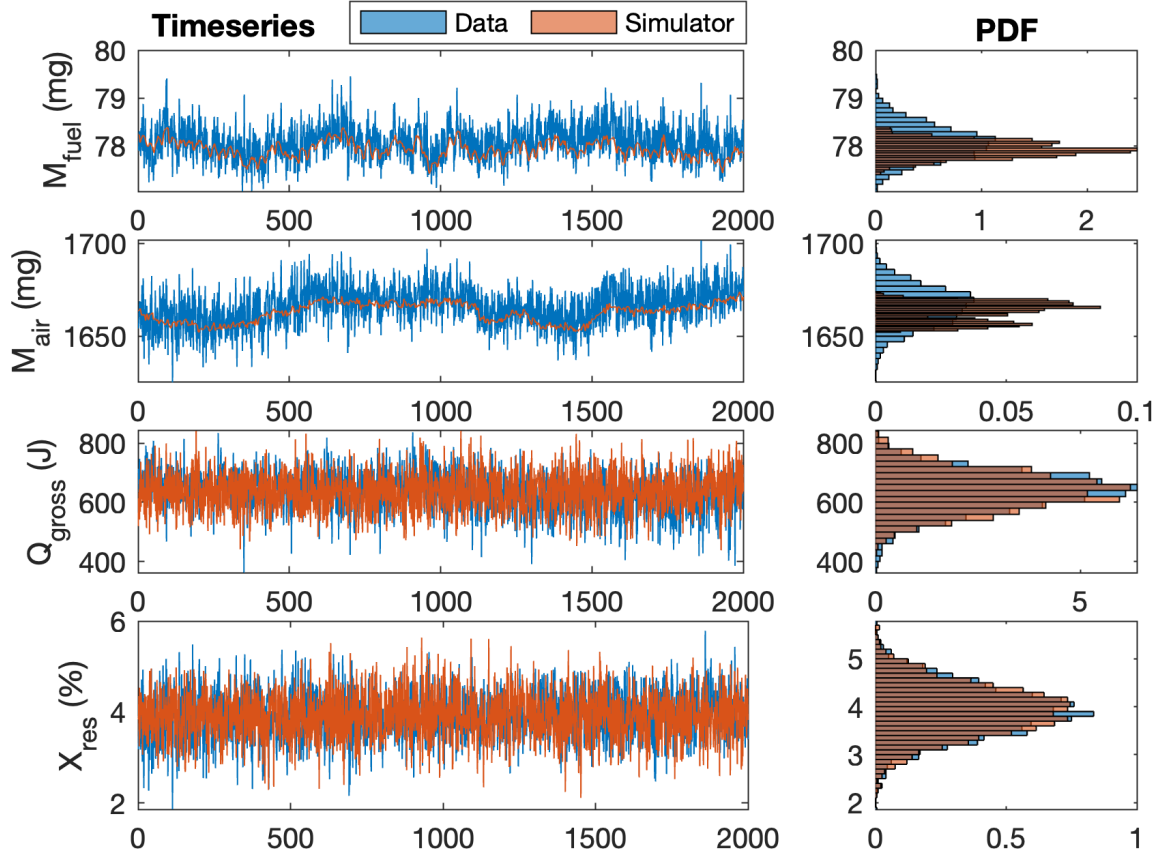


Figure 5.2: Comparison between experimental and simulated values for the physics-based model at 7.4 mg of diesel quantity and 42 deg dTDC injection timing.

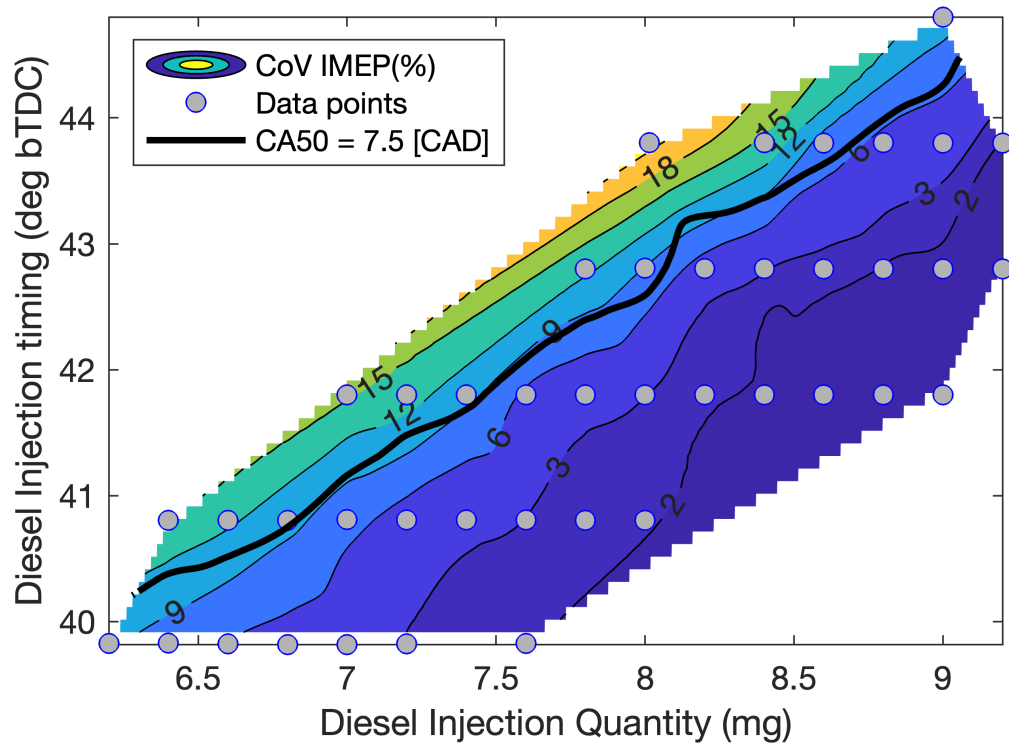


Figure 5.3: Experimental conditions at an average NH_3 replacement rate of 78.9%.

system, they do not necessarily need to correspond to specific physical values or phenomena. As such, it may be possible for certain seemingly-arbitrary parameters to capture the total effect of many interacting phenomena as a smaller number of summarizing parameters. We defined three such parameters— ϕ_d , ϕ_{NH_3} , and ϕ_{air} —which function as scalar multipliers applied to the observed diesel, ammonia, and air masses. However, in testing, we found that the latter two had virtually no effect, so we discarded them to reduce the size of our parameter space and allow for more fine-grained exploration of the singular ϕ_d parameter with shorter training times for pretrained batches. This parameter was bounded between 0.85 and 1.3, corresponding to the range of expected values for diesel observations. In testing, the Sim Modifier was able to predict the correct value on a second instance of the simulator within an error range of ± 0.03 in 5 minutes. However, it struggled to converge consistently on the real engine, as discussed in the results section of this chapter.

5.3 Hardware Infrastructure

The hardware and firmware infrastructure, FireBox, hosts the online learning software and implements the interface to the Engine Control Unit (ECU) and the spiking neural network processors. The system is composed of the widely available Raspberry Pi 5 and BeagleV-Fire single-board computers. The Raspberry Pi 5 provides a quad-core Arm Cortex A76 application class processor ideal for running CPU-based training and simulation software in an embedded form factor. The BeagleV-Fire features the Microchip PolarFire SoC which provides a real-time capable RISC-V SoC FPGA suitable for control and communications tasks. We refer to the BeagleV-Fire and Raspberry Pi 5 hardware as the real-time and application domains respectively. The hardware system is shown in Fig. 5.4 with the application hardware on the right and the real-time hardware on the left.

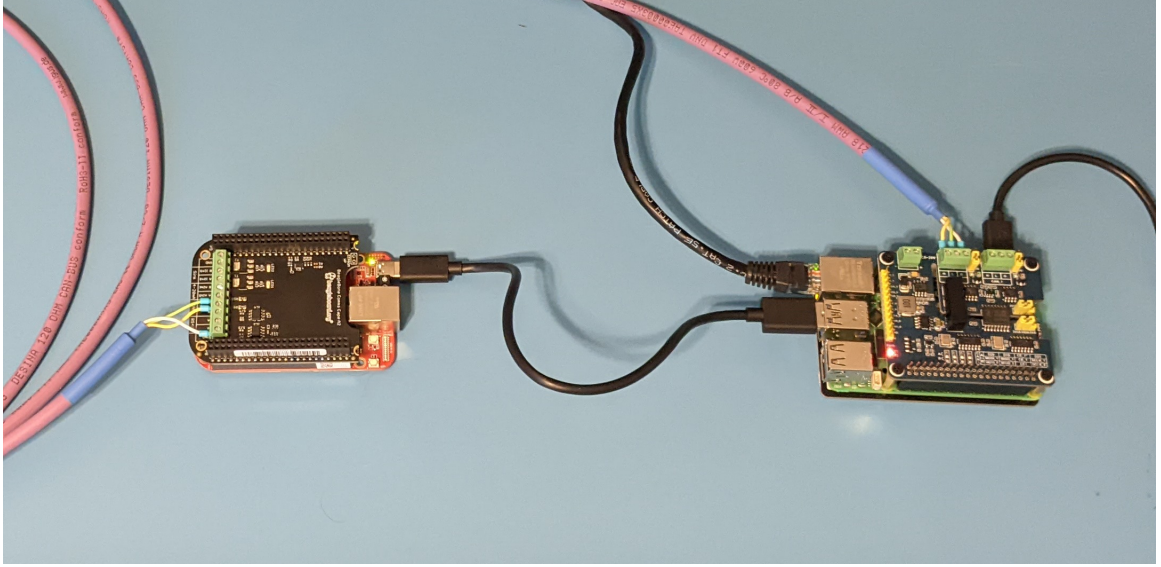


Figure 5.4: The FireBox hardware configured for co-simulation with hardware-in-the-loop and software diesel engine simulator.

This project uses multiple instances of the μ Caspian neuromorphic processor implemented in the PolarFire low-power FPGA fabric (19). Additional control and framing logic is also implemented in the FPGA to support an efficient memory-mapped interface between the neuromorphic coprocessors and the RISC-V host processor. A controller area network (CAN) and transceiver enables the real-time processor to communicate with the ECU and other sensors. A custom embedded Linux image includes a userland system service and a kernel device driver to implement control of multiple neuro-processor cores. Various message queuing patterns are used to communicate over Ethernet or USB between the real-time and application domain hardware allowing one of the neuro-processor cores to control the engine while the others are pre-loaded with new networks as required by the online learning algorithm. In addition, a co-simulation setup was developed that combines CAN communications, the hardware framework, and the training software to support integration testing and continued algorithm work when access to the diesel engine testing room is not practical or unavailable. The co-simulator can optionally run

the neuromorphic core simulator used for training in lock-step with the hardware implementation for functional verification.

5.3.1 Hardware Infrastructure Performance

The hardware infrastructure developed for this project attempts to provide a convenient development platform for both hardware and software teams while simultaneously supporting low-level and real-time hardware control requirements. To support this aim, we split the hardware infrastructure into application and real-time computing domains. This section presents results for key capabilities of the real-time hardware platform leaving the application computing hardware.

Two metrics of interest for real-time control and online learning applications are inference and network loading time of the neuromorphic coprocessors. A diagram of the firmware and FPGA logic implementation is shown in Fig. 5.5. The high-priority controller software thread handles inference by encoding real-valued engine states into discrete spikes, issuing commands to the memory-mapped neuromorphic coprocessor, and finally decoding spikes into continuous control values. The lower priority dispatcher thread handles requests to load networks on inactive coprocessors and swapping which coprocessor is controlling the engine. Low-latency and deterministic inference time is important to meet the deadlines for real-time engine control. In addition, a reasonably fast and reliable network loading time allows the online learning infrastructure to have a responsive real-world interface that can load a new network as soon as the next engine cycle.

The average inference time and network loading time for the trained networks used in testing is shown in Fig. 5.6. The external communications overhead is neglected for this analysis, but experimentally the controller was easily able to meet the 50 ms periodic deadline for real-time control of the diesel engine. The inference time begins when the engine state is received from the ECU until after the control spikes are decoded. Typical inference times range from 2 to 4 ms and are dependent on the

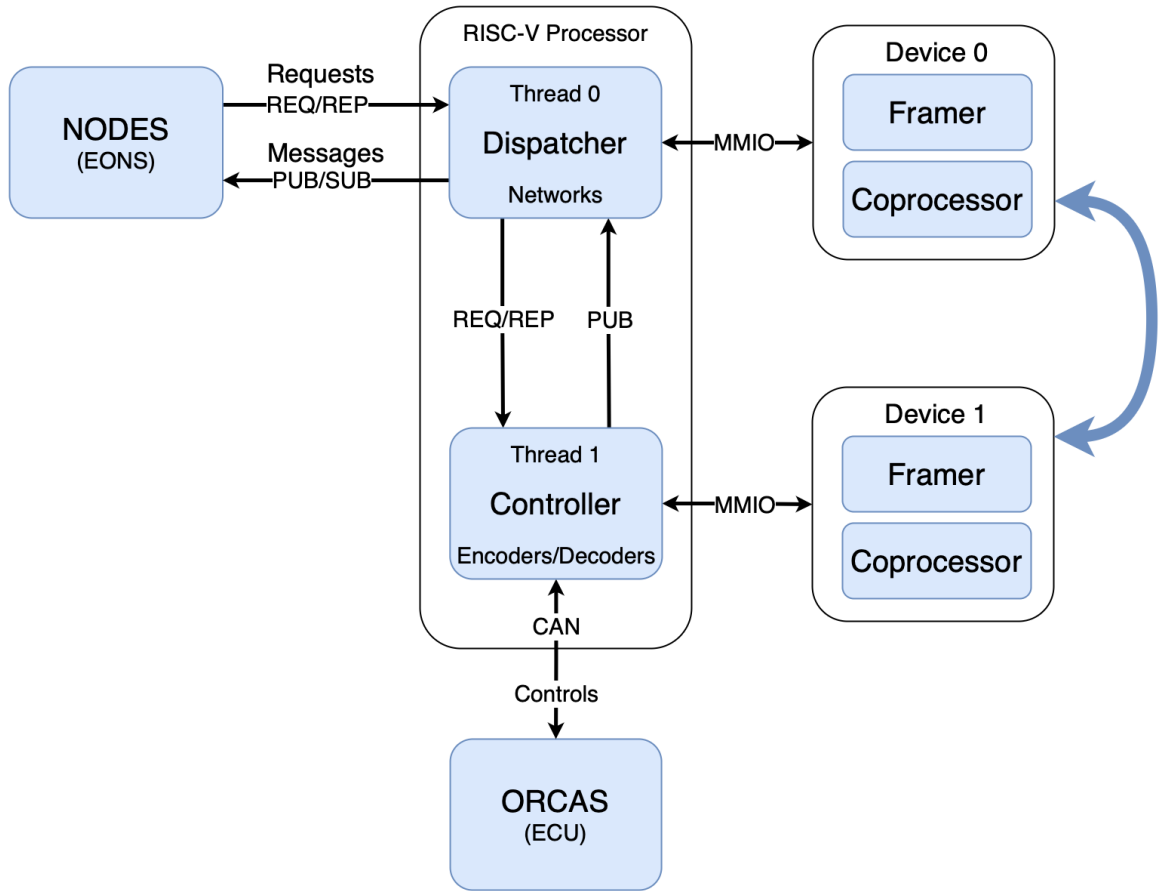


Figure 5.5: A diagram of the hardware and software services implemented on the FireBox neuromorphic engine controller.

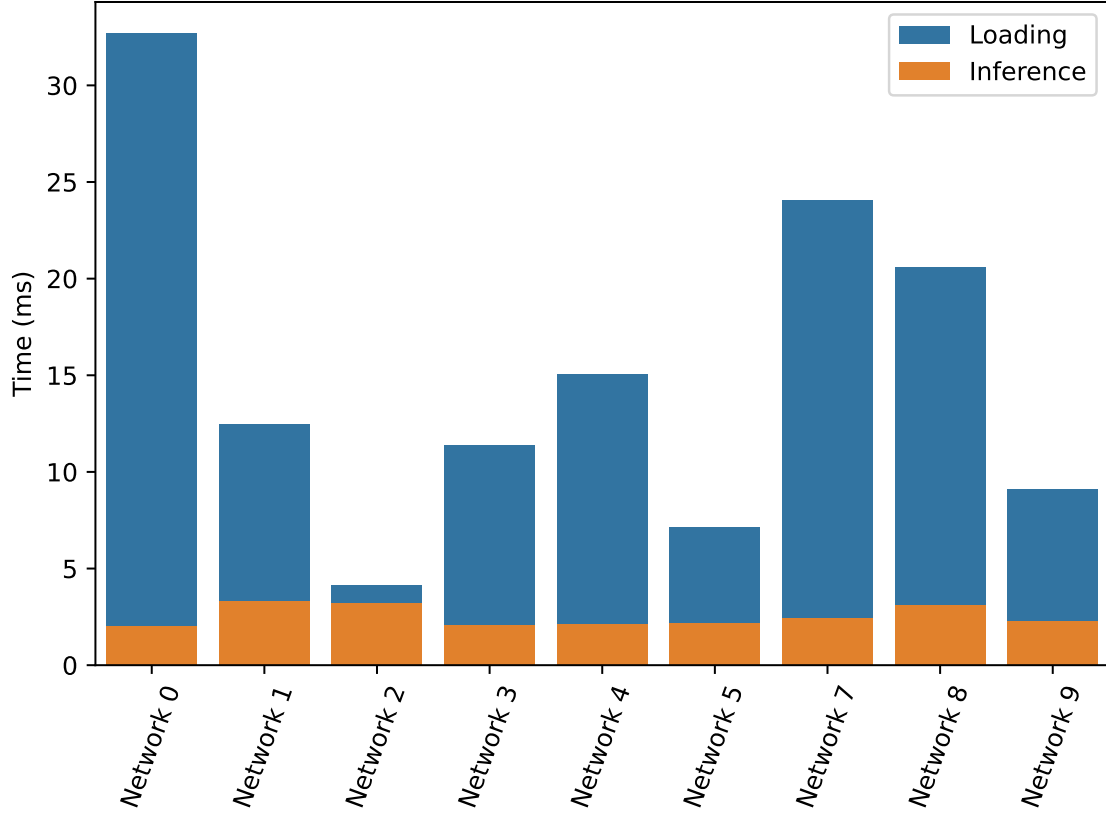


Figure 5.6: Average network loading time and inference time for various trained networks on the FireBox neuromorphic engine controller.

network parameters and spiking activity. This also includes a minimum 1 ms worst-case execution time due to a limitation in the implementation which requires a wait state to ensure that all spike outputs have been received. The network loading time varies with the size of the network but is typically between 5 and 30 ms.

5.4 Results

Due to time limitations, after finalizing all of the cost function coefficients and simulation parameters, we were only able to record one testing run on the physical engine. Due to file corruption, some of the data from that testing run was lost. However, the period of data that we retained is very informative. Using a pretrained

leaderboard was necessary for this problem, as training networks with the engine simulator can take 10 minutes or longer. Fig 5.7 shows the cost over time and relevant values received from both the engine and the NODES framework.

At the beginning of the test, the simulation parameter ϕ_d is at its initial value of 1.0, and the startup network is slightly increasing the amount of diesel from the base value. While this is not the desired behavior, it does allow the engine to be stable and efficient. From the real-world observations, the Sim Modifier interface first predicts a diesel observation multiplier of 1.145, and a network which performed well in simulation on that parameter is deployed. However, this network exclusively cuts diesel as low as it can every timestep. This dramatically reduces combustion efficiency to 70%, while pushing the CA_{50} and coefficient of variance far outside of their desired range. As the system continues to inject too little diesel, the engine state continues to worsen, to the point where we nearly had to shut it off to avoid rendering it non-operational. However, the next parameter update from the Sim Modifier was received just before that was necessary, lowering ϕ_d back down to 1.015. In response, the Coordinator deployed a network which had the exact opposite behavior, exclusively injecting as much diesel as possible. Although not cost effective, this undid the damage done by the previous network and brought the engine back to a stable state. Finally, the last parameter update set ϕ_d to 1.185, the highest value yet, but the network that was fetched performed exactly as we intended. It slightly reduced the amount of diesel injected, while minimally impacting the performance of the engine.

The clear explanation for these observations is that the simulator simply cannot consistently capture the behavior of the real engine. Although the final network reduced diesel injection mass as we expected from a network trained on a high positive diesel observation multiplier, the other networks did not adhere to this logic. They scored well on the engine simulator, but did not perform well on the real engine. Furthermore, the incapability of the simulator to accurately mimic the physical engine prevented the Sim Modifier from functioning correctly, a problem that is exacerbated

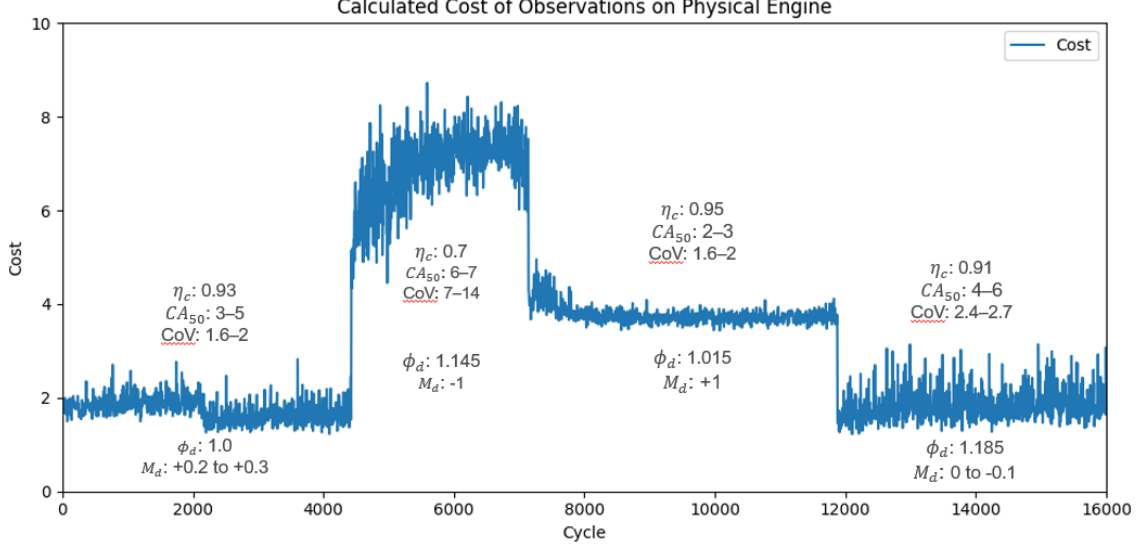


Figure 5.7: Plot of cost over a 30-minute interval calculated from observations on the physical engine while being controlled by NODES. Annotations indicate observation, action, and parameter values.

by the nature of the engine system as a feedback loop that strays further from the behavior expected by the data-driven model the longer it is made to behave unusually.

This test demonstrates the limitations of the NODES framework. In order for it to function properly, it is crucial that the simulated environment be similar enough to the real one that there exist some parameter values for which a network trained on those values can be expected to perform reasonably well when deployed on the real system. Furthermore, a simulated environment that is governed by entirely different mechanisms than the real world will result in the Bayesian optimization algorithm being unable to converge upon parameters which best mimic the real world, as the shape of the loss function in parameter space will be chaotic and unpredictable.

As a result of this test, we implemented the loss threshold mechanism on the Sim Modifier and the Real-World Leaderboard as described in Chapter 3. The former helps avoid scenarios where the Sim Modifier sends a parameter update despite failing to find any simulator configuration which adequately mimics the real system’s behavior. The latter allows us to avoid deploying networks that have previously been

demonstrated to perform terribly on the real system, so that when erratic behavior does occur, NODES is able to learn from it and avoid it in the future.

Chapter 6

Conclusion

In this work, we presented NODES, a novel framework for adaptive real-time control of dynamic physical systems using spiking neural networks. The framework was implemented by Ethan Maness, Karan Patel, and Tyler Nitzsche and tested with the help of ORNL collaborators Bryan Maldonado, Brian Kaul, Brett Witherspoon, and Aaron Young, and funding provided by the UT-ORII seed program. Through two case studies we demonstrated the capabilities of the framework, as well as its limitations, illustrating its potential for general use in other applications.

The first case study illustrates ideal performance in an ideal scenario. The desired adaptive behavior is observed in response to changes in the environment, and when run with a pretrained leaderboard, the framework performs very consistently. Fast training times, parameter optimization times, and a simple problem allow NODES to function exactly as intended. In contrast, the second case study shows the factors which can inhibit the framework’s performance. Exceedingly complex systems cannot be adequately simulated with the necessary speed on the limited resources of an embedded chip, preventing the framework from being able to track and adjust for current real-world conditions.

Initially, we expected that the key limiting factors would be the rate of network training and changing conditions, as well as the sensitivity of chaotic systems to slight

deviations in those conditions. These expectations proved to be mostly accurate, although we were able to circumvent slow online training by supplying the framework with a batch of networks already trained offline beforehand.

There essentially exists a boundary plane dividing the problems which NODES can and cannot adequately handle, defined by its three key limiting factors—complexity of the physical system, sensitivity of parameters to deviations, and the rate of change of the environmental conditions. As one of these metrics increases, the other two must decrease for a problem to remain solvable by NODES. For example, NODES may perform well on a sufficiently simple system despite it having very sensitive parameters and a rapid rate of change, because a simple system can be simulated accurately and quickly, allowing the Sim Modifier interface to reverse engineer the real-world conditions with high precision and speed while the EONS algorithm rapidly trains new networks on the reverse engineered parameters to keep up with a high rate of change. However, if a system is very complex, it must change slowly and the simulator must be able to adequately approximate its behavior even with slightly incorrect parameters.

Some of these issues may be addressed through further explorations. The limitation on simulation complexity can be loosened by outsourcing the simulation process to a remote computing cluster, either locally or through the cloud. This would significantly improve the scope of problems on which NODES is capable of achieving good performance. Other optimization algorithms may be explored as well, as different algorithms have their own strengths and weaknesses. In this work, we used Bayesian optimization as it tends to perform well on functions which are expensive to compute, but it is possible that other algorithms may perform better for certain applications. Planned developments for the near future include testing the NODES framework further on different classes of problems to learn more about its capabilities, as well as constructing a physical Polebalance system to compare its performance on a real system versus a simulator. We believe that NODES shows potential for a great breadth of use cases and further improvements, as is often the

case for such exploratory research. Neuromorphic computing is a promising field with unique strengths that allow it to excel in scenarios where other solutions fail, and nascent developments such as this work are the stepping stones towards unearthing such potential.

Bibliography

- [1] C. D. Schuman, C. P. Rizzo, G. S. Rose, and J. S. Plank, “Embracing the hairball: An investigation of recurrence in spiking neural networks for control,” in *2024 Neuro Inspired Computational Elements Conference (NICE)*, pp. 1–5, 2024. [2](#)
- [2] P. Chandarana, M. Mohammadi, J. Seekings, and R. Zand, “Energy-Efficient Deployment of Machine Learning Workloads on Neuromorphic Hardware ,” in *2022 IEEE 13th International Green and Sustainable Computing Conference (IGSC)*, (Los Alamitos, CA, USA), pp. 1–7, IEEE Computer Society, Oct. 2022. [2](#)
- [3] G. N. Katsaros and K. Nikitopoulos, “Toward fully neuromorphic receivers for ultra-power efficient communications,” 2025. [2](#)
- [4] N. Pavlidis, O. Tasoulis, V. Plagianakos, G. Nikiforidis, and M. Vrahatis, “Spiking neural network training using evolutionary algorithms,” in *Proceedings. 2005 IEEE International Joint Conference on Neural Networks, 2005.*, vol. 4, pp. 2190–2194 vol. 4, 2005. [7](#)
- [5] R. Storn and K. Price, “Differential evolution – a simple and efficient heuristic for global optimization over continuous spaces,” *Journal of Global Optimization*, 1997. [7](#)
- [6] A. Belatreche, L. P. Maguire, M. McGinnity, and Q. X. Wu, “Evolutionary design of spiking neural networks,” *New Mathematics and Natural Computation*, vol. 02, no. 03, pp. 237–253, 2006. [7](#)

- [7] G. López-Vázquez, M. Ornelas-Rodriguez, A. Espinal, J. A. Soria-Alcaraz, A. Rojas-Domínguez, H. J. Puga-Soberanes, J. M. Carpio, and H. Rostro-Gonzalez, “Evolutionary spiking neural networks for solving supervised classification problems,” *Computational Intelligence and Neuroscience*, vol. 2019, 2019. [7](#)
- [8] I. Daza, R. Izquierdo, L. Martínez, O. Benderius, and D. F. Llorca, “Sim-to-real transfer and reality gap modeling in model predictive control for autonomous driving,” *Applied Intelligence*, 2022. [8](#)
- [9] J. Collins, R. Brown, J. Leitner, and D. Howard, “Traversing the reality gap via simulator tuning,” 2020. [8](#)
- [10] A. Ligot and M. Birattari, “On mimicking the effects of the reality gap with simulation-only experiments,” in *Swarm Intelligence* (M. Dorigo, M. Birattari, C. Blum, A. L. Christensen, A. Reina, and V. Trianni, eds.), (Cham), pp. 109–122, Springer International Publishing, 2018. [8](#)
- [11] H. Qiu, M. Garratt, D. Howard, and S. Anavatti, “Towards crossing the reality gap with evolved plastic neurocontrollers,” in *Proceedings of the 2020 Genetic and Evolutionary Computation Conference, GECCO ’20*, (New York, NY, USA), p. 130–138, Association for Computing Machinery, 2020. [8](#)
- [12] C. D. Schuman, J. S. Plank, A. Disney, and J. Reynolds, “An evolutionary optimization framework for neural networks and neuromorphic architectures,” in *2016 International Joint Conference on Neural Networks (IJCNN)*, pp. 145–154, IEEE, 2016. [11](#)
- [13] C. D. Schuman, J. P. Mitchell, R. M. Patton, T. E. Potok, and J. S. Plank, “Evolutionary optimization for neuromorphic systems,” in *Proceedings of the 2020 Annual Neuro-Inspired Computational Elements Workshop*, pp. 1–9, 2020. [11](#)

- [14] J. S. Plank, C. P. Rizzo, C. A. White, and C. D. Schuman, “The cart-pole application as a benchmark for neuromorphic computing,” *Journal of Low Power Electronics and Applications*, vol. 15, no. 1, pp. 1–27, 2025. [23](#)
- [15] P. E. K. Donaldson, “Error decorrelation: a technique for matching a class of functions,” in *Proceedings of the Third International Conference on Medical Electronics*, pp. 173–178, 1960. [23](#)
- [16] J. S. Plank, K. E. M. Dent, B. Gullett, C. P. Rizzo, and C. D. Schuman, “The RISP neuroprocessor – open source support for embedded neuromorphic computing,” in *IEEE International Conference on Rebooting Computing (ICRC)*, (San Diego), December 2024. [23](#)
- [17] K. Patel, E. Maness, T. Nitzsche, E. G. Brown, B. Witherspoon, A. Young, B. Maldonado, B. Kaul, J. S. Plank, and C. D. Schuman, “Evolution at the edge: Real-time evolution for neuromorphic engine control,” in *2025 Neuro Inspired Computational Elements (NICE)*, pp. 1–8, 2025. [31](#), [34](#)
- [18] C. D. Schuman, S. R. Young, B. P. Maldonado, and B. C. Kaul, “Real-time evolution and deployment of neuromorphic computing at the edge,” in *12th International Green and Sustainable Computing Conference (IGSC)*, pp. 1–8, IEEE, 2021. [31](#)
- [19] J. P. Mitchell, C. D. Schuman, R. M. Patton, and T. E. Potok, “Caspian: A neuromorphic development platform,” in *Proceedings of the 2020 Annual Neuro-Inspired Computational Elements Workshop*, pp. 1–6, 2020. [38](#)

Vita

Ethan Maness was born and raised in Kingsport, Tennessee, alongside his sister Samantha Maness. His parents, Karen and Michael Maness, supported his academic career. Ethan developed an interest in computer programming at a young age through a hobby of modifying video games, and his experience on a FIRST Robotics Competition team solidified Computer Science as his desired field of study. Although he worked at multiple software engineering internships during his undergraduate education, he chose to pursue a graduate degree after taking a Biologically-Inspired Computing course taught by Catherine Schuman, who proceeded to take him on as an undergraduate research assistant, and then a graduate research assistant, where he studied neuromorphic computing. He graduated with his Master of Science in Computer Science in December 2025.