Reservoir Computing in an Evolutionary Neuromorphic Framework

A Dissertation Presented for the

Doctor of Philosophy

Degree

The University of Tennessee, Knoxville

John Jeffrey McClain Reynolds

December 2019

© by John Jeffrey McClain Reynolds, 2019
All Rights Reserved.

Acknowledgments

I want to thank Dr. James S. Plank and Dr. Catherine D. Schuman, for the guidance they have afforded me from my undergraduate to my doctoral studies. I would also like to thank the committee members of this thesis: Dr. Mark E. Dean, Dr. Garret S. Rose, and Dr. Andy Sarles for providing insightful and constructive recommendations. Finally, I would like to thank my loving wife, Kristen. The unwavering support she provided throughout this period of my life has been invaluable.

This work includes previously published articles from both ACM and IEEE. The digital object identifier (doi) is 10.1145/3229884.3229896 for the paper titled "A Comparison of Neuromorphic Classification Tasks," which is utilized heavily in chapter 3. The paper "Intelligent Reservoir Generation for Liquid State Machines using Evolutionary Optimization" is reproduced in chapter 4, and can be found with the digital object identifier 10.1109/IJCNN.2019.8852472. The International Joint Conference on Neural Networks in IEEE published the paper alongside conference proceedings. The ACM and IEEE have both provided permission to reproduce the publications mentioned above as a part of this dissertation work.

Abstract

Neuromorphic computing is an emerging hardware paradigm for doing non-traditional computing. It has advantages over typical von Neumann systems in a myriad of different situations. In particular, it offers attractive power savings over traditional hardware, by doing spiking neural network computations. However, programming a neuromorphic spiking system is very challenging, and thus an active field of research. This work explores using the TENNLab group's neuromorphic computing framework with reservoir computing, a method for utilizing either spiking or non-spiking neural networks as dynamical systems (called reservoirs) to filter and map information from one dimension to another to form useful intermediate data representations. In this case, spiking recurrent neural networks are used to do the processing. We delve into creating reservoirs with evolutionary genetic algorithms, and we explore parameters and applications across the available TENNLab architectures.

Table of Contents

1 Introduction 1

1.1 Overview . . . 1

1.2 Artificial Neural Networks . . . 2

1.3 Spiking Neural Networks . . . 5

1.3.1 NIDA . . . 6

1.3.2 DANNA2 . . . 7

1.3.3 mrDANNA . . . 8

1.4 Evolutionary Optimization of Neuromorphic Systems . . . 8

1.5 Reservoir Computing . . . 9

1.6 TENNLab Software Structure . . . 14

1.7 Dissertation Structure . . . 16

2 Related Work 18

2.1 Overview . . . 18

2.2 Spiking Recurrent Neural Networks . . . 18

2.3 Reservoir Computing . . . 20

2.3.1 Liquid State Machines . . . 20

2.3.2 Echo State Networks . . . 21

2.4 Genetic Algorithms . . . 22

2.5 Reinforcement Learning . . . 23

3 A Comparison of Neuromorphic Classification Tasks 24

3.1 Overview . . . 24

3.2 Neural Networks and Machine Learning Algorithms 26
3.2.1 Deep Learning Approaches 26
3.2.2 Spiking Recurrent Neural Networks 28
3.2.3 EONS: Evolutionary Optimization of Neuromorphic Systems . 29
3.2.4 Reservoir Computing 30
3.2.5 Summary of Neural Networks and Machine Learning Algorithms 31
3.3 Classification Applications 32
3.3.1 Iris, Breast Cancer, Pima 32
3.3.2 Satellite Radio 33
3.3.3 Electroencephalogram (**EEG**) 33
3.3.4 Consonants vs. Vowels 34
3.3.5 Summary of Data Sets 35
3.4 Experimental Setup 35
3.5 Results 36
3.6 Conclusion 40

4 Reservoir Computing and Genetic Algorithms in the TENNLab Framework 41
4.1 Overview 41
4.1.1 Spiking Model 42
4.1.2 Training: Evolutionary Optimization of Neuromorphic Systems 42
4.1.3 EONS Applied to LSMs 43
4.2 Experimental Results 43
4.2.1 Ionosphere 44
4.2.2 Experiment Design 44
4.2.3 Grid Search Results 45
4.3 Evolutionary Optimization of Reservoir Fitness 51
4.4 Discussion 53
4.5 Conclusion 57

5 Reservoir Computing for LIDAR Control Applications 58
5.1 Overview 58

5.2 Reinforcement Learning . . . 59
5.3 Policy Gradients . . . 61
5.4 TENNLab LIDAR Control Applications . . . 63
5.4.1 Asteroids . . . 63
5.4.2 Spaceinvaders . . . 64
5.4.3 Bowman . . . 65
5.5 Experiment Design . . . 65
5.6 Results . . . 68
5.6.1 Asteroids . . . 68
5.6.2 Spaceinvaders . . . 71
5.6.3 Bowman . . . 73
5.7 Discussion and Conclusion . . . 76

6 Future Directions & Conclusions 78

Bibliography 80

Appendix 97
A Readout Layer . . . 98
A.1 Readout Class . . . 98
A.2 Readout Layer Functions . . . 101

Vita 110

List of Tables

3.1 Summary of the neural networks and machine learning algorithms employed in this paper. 31

3.2 Summary of classification data sets and their parameters as they affect the various machine learning algorithms. N, which corresponds approximately to spiking neural network run-time, applies only to EONS and RC, and M, which corresponds to when output pulses are counted, applies only to RC. 36

3.3 Percentage accuracies of each model on each data set. Each number presents the best classification accuracy on the testing set of data. . . 37

3.4 Neuron and synapse counts for the networks that produced the classification accuracies in Table 3.3. 39

4.1 Correlation matrix of associated recorded reservoir variables. 52

List of Figures

1.1 Diagram of reservoir computing. Input is passed into the reservoir (liquid in this case), then after a predetermined amount of time, the reservoir is sampled and the sample is passed into the readout layer as a feature vector. The readout layer is then trained on the features taken from the reservoir. 12

1.2 This is an example of how the counting kernel causes separation in a reservoir. The classes are the three categories found in the well-known Fisher's Iris dataset [28]. The number of fires is clearly distinguishable between the three examples. 15

3.1 Two example snippets from the **Radio** dataset: 8PSK on the top, and GFSK on the bottom. Real values are plotted in black, and imaginary values in red. 33

3.2 Example time series plots of two EEG signals: healthy (orange), and epileptic(blue). 34

3.3 Spectrogram of MFCC features from a spoken phoneme 'iy'; a vowel. Each column is a MFCC vector with 13 values, calculated every 10 ms over 25 ms of samples. 35

3.4 The top figure is a bar graph of model accuracy per data set. Accuracies are from table 3.3. The bottom figure is a bar graph of network sizes for comparison. Network sizes are in table 3.4 38

4.1 The heat maps above were generated from a coarse-grained grid search on randomly created reservoirs trained on the ionosphere dataset. Each block in the heat map represents the best accuracy from a population of 100. The number of hidden neurons and the synaptic density are varied on the y and x axes respectively. 46

4.2 The heat maps above were generated from fine-grained grid search on randomly created reservoirs trained on the ionosphere dataset. Each block in the heat map represents the best accuracy from a population of 100. The number of hidden neurons and the synaptic density are varied on the y and x axes respectively. In particular, the fine-grained search is across 60 neurons centered on the best accuracy from the top level grid search: 0.15 synaptic density. This demonstrates that the hyperparameter configuration is an integral part of reservoir success. . 47

4.3 The histograms depicted above represent the accuracy distribution across configurations utilized in the grid search. It is clear that a valid choice of hyperparameters still does not ensure successful reservoirs. Further, even within a fine-grained search across a good choice of hyperparameters, many liquid evaluations may be required. The accuracy distribution is higher on average in the fine-grained approach, but the highest performing reservoirs are still a small portion of the overall distribution. 49

4.4 These graphs are generated from all 60k of the grid search tests. The average silhouette coefficient increases with accuracy, but still has a wide range of values. The spectral radius also narrows to a certain range with higher accuracy, but spectral radius values in that range exist across a majority of the spectrum of accuracy, so having a value in that range does not guarantee success, nor prevent it. 50

4.5 Correlation matrix of variables recorded throughout the reservoir generation process. 52

4.6 Top 500 performing networks from each of the methods. The y-axis is presented on a log scale. The plots demonstrate that the EONS results are skewed towards higher performance. Overfitting can be remedied through manual tweaking of parameters, and is less apparent when reviewing more than the top 500 performers. 54

4.7 The top two figures are hex-bin plots of the EONS tests performed. The top-left plot is epoch 0, and the top-right plot is epoch 11. Each plot is of all 50 networks from all 100 runs for a total of 5k. Each hexagon represents a bin, and all of the samples falling into that region are integrated into that bin. The color gradient represents the number of samples in a bin. EONS converges to the expected accuracy (approx. 90%) regardless of the starting configuration, which is an inherently different result than the two plots at the bottom; the left being the coarse-grained grid search, and the right being of all 5 fine-grained grid searches. 55

4.8 The top two figures are hex-bin plots of the EONS tests performed. The top-left plot is epoch 0, and the top-right plot is epoch 11. Each plot is of all 50 networks from all 100 runs for a total of 5k. Each hexagon represents a bin, and all of the samples falling into that region are integrated into that bin. The color gradient represents the number of samples in a bin. EONS converges to the expected accuracy (approx. 90%) regardless of the starting configuration, which is an inherently different result than the two plots at the bottom; the left being the coarse-grained grid search, and the right being of all 5 fine-grained grid searches. 56

5.1 An example screenshot of the TENNLab Asteroids simulation. The LIDAR sensors are represented by dotted lines emanating from the player's ship. The color of the line represents the distance of a detected object. 64

5.2 An example screenshot of the TENNLab Spaceinvaders simulation. The LIDAR sensors are represented by lines extruding from the player's ship. The color of the line represents the distance of a detected object. 65
5.3 An example screenshot of the TENNLab Bowman simulation. The LIDAR sensors are represented by lines emanating from the player's ship. The line turns white when and object is detected. 66
5.4 Polebalance results readout only. The top is of the readout trained on the TENNLab application, and the bottom is trained on OpenAI gym's implementation. The success serves as a baseline for policy gradient success without a reservoir. 67
5.5 Trajectories of individual EONS runs for Asteroids 69
5.6 The best network taken from the EONS run was trained for an extended period of time. The graphs above show how behavior changed throughout they training epochs. 70
5.7 These heatmaps demonstrate the accuracy the best network throughout training time. 72
5.8 Relative distribution of output events for Asteroids while training the readout. 73
5.9 The EONS trajectories for Spaceinvaders. 74
5.10 Training the readout layer for Spaceinvaders, both without a reservoir and with the best reservoir produced by EONS. 74
5.11 The EONS trajectories for Bowman. 75
5.12 Training the readout layer for Bowman, both without a reservoir and with the best reservoir produced by EONS. 76

A.1 These are the includes required for this code to compile. Several of the functions found in the following code will not work and will throw compiler errors if these includes have not been place at the top. . . . 98
A.2 This is the randomization function used to initialize neural network weights in the readout layer / multinomial logistic regression. 98

A.3 Readout class. 99
A.4 This is the readout class constructor. 100
A.5 This is the softmax function. Simply pass a value of 1 for temperature in order to have a regular softmax function. 101
A.6 This is the forward function. It propagates the input data vector forward by matrix multiplying with the network weights. Call this before calling the backwards function. 102
A.7 This is the backwards pass of the neural network. Must be called after the forward pass. It updates both the regular weights and the bias weights. 103
A.8 This creates a diagonal matrix from the passed in vector. 104
A.9 This performs a matrix multiply. 104
A.10 This function implements the REINFORCE policy gradients algorithm. 105
A.11 This function implements the gradient computation based on the softmax function. This is for use with policy gradients. 106
A.12 This function implements the computation of gradients based on the derivative of cross-entropy loss. This is used in classification problems. 107
A.13 This function implements part 1 of gradient checking. This ensures the gradient checking computes the numerical limit formulation of a gradient in order to compare against the derived version. 108
A.14 This function implements part 2 of gradient checking. This ensures the gradient checking computes the numerical limit formulation of a gradient in order to compare against the derived version. 109

Chapter 1

Introduction

1.1 Overview

The study of neuromorphic computing was born in the late 1980s from Carver Mead's idea to create circuitry modeling the capabilities of neural networks found in biological systems [74]. The field has recently begun to gain steam with the realization of the slowing of Moore's Law's [135]. Speed improvements born from Intel co-founder Gordon Moore's prediction have waned. Researchers are exploring viable alternatives for continuing progress in computing. Neuromorphic computing is particularly exciting given recent advancements found in neural networks through the many different available mathematical and neurological models. However, at the time of writing this dissertation, training spiking neural networks (SNNs) to perform complex tasks is difficult, and far from completely understood. Many potential models for doing neuromorphic computing remain unexplored. It is within this context that the TENNLab Neuromorphic Research group operates. We explore both models and learning paradigms in hardware and software. We study the application of these systems to practical problems in hopes of shedding light on viable use cases of (SNNs) for the field of machine learning.

The TENNLab research group is composed of hardware, software, and applications. We are dedicated to the study of these neuromorphic systems while applying

them to informative applications. We have several different models of neuromorphic architectures, each with various properties and constraints. NIDA, DANNA2, and mrDANNA are three prominent models of the research group [90]. Historically, the group has focused on a genetic algorithm known as evolutionary optimization of neuromorphic systems (EONS) for training these neural networks [108]. There are many reasons genetic algorithms are a particularly enticing method for training these systems. In specific, they avoid gradient computation and provide nice convergence properties, whereas gradient-based methods can get stuck in local minima.

This dissertation, however, is focused on ushering in a new paradigm for utilizing these systems: reservoir computing (RC) [132]. In this work, we discuss the differences between the training methodologies EONS and RC. We also compare and contrast the usage of each on several different tasks. Further, we empirically explore the successful parameterization of our architectures as RC systems.

1.2 Artificial Neural Networks

Artificial neural networks are mathematical models based loosely on the brain [49]. Artificial neural networks mainly consist of only two structures: neurons and synapses. Neurons perform functions (called activation/transfer functions), and synapses represent a weighted dataflow between the neuronal functions. A simple example of this is that of a perceptron. In perceptrons, the neuron's function is simply a linear combination of inputs, and their related synapses weight the inputs. The following simple mathematical functions describe the perceptron:

$$o = f(\sum_{k=1}^{n} x_k \cdot W_k) \tag{1.1}$$

$$f = \begin{cases} 1 & w \cdot x + b > 0, \\ 0 & otherwise \end{cases} \tag{1.2}$$

The idea of artificial neural networks (a form of function composition) has been around since the mid 20th century. The first recorded documentation of artificial neural networks comes from Warren McCulloch and Walter Pitts in 1943 [73]. Their neuron model (known as the McCulloch-Pitts model) is straightforward, only requiring the aggregation of inputs, whose sum passes through a decision-making function. Another breakthrough was that of Hebbian Learning, introduced by Donald O. Hebb in 1949 [41]. Hebbian learning is an extrapolation of synaptic plasticity seen in real biological neural networks. Frank Rosenblatt invented the perceptron in 1958 [98]. Marvin Minsky and Seymour Papert later explored shortcomings, notably the perceptron's inability to compute XOR [75], and further citing that using multiple layers with non-linear neuronal functions (referred to as activation or transfer functions) may be capable of fixing that shortcoming. Hence, researchers moved towards studying multilayer perceptrons. However, simple learning rules were no longer viable. Large multilayer models required the derivation of more complex solutions for training. Backpropagation, a method for teaching neural networks to solve problems, was introduced by many different scientists in several contexts, from control theory to dynamic programming and optimization [42]. The basis for modern artificial intelligence stems from these important developments. However, in modern times, references to artificial neural networks are typically discussed within the context of larger networks known as deep neural networks.

Deep Learning [58] is a term incorporating the previously mentioned ideas into a broader framework differentiated primarily by the "depth" of the network (number of layers). GPUs are frequently used to train deep neural networks on vast amounts of data because of their efficient implementations of hardware-level matrix multiplication. More extensive networks require more linear operations than shorter networks in training. Perhaps the most popular method for training neural networks is known as gradient descent [100]. Gradient descent requires optimizing weights by taking the partial derivative of a loss function with respect to network weights. A loss function describes a cost associated with inputs (in this case, network outputs). The goal is to minimize the cost/loss associated with outputs. Minimizing the loss is

equivalent to minimizing the amount of error associated with a given problem one is attempting to model. Therefore, taking the partial derivative of a loss function with respect to the synaptic weights gives us information describing the rate of change in the error with respect to individual weights. Gradient descent uses this information to backward propagate (backpropagation) the gradient from the loss to the respective weights. The loss function descends such that the error associated with individual weights lessens, hence the term gradient *descent*. An example loss function is that of the cross-entropy loss (shown in the below equation).

$$-\sum_{k=1}^{n} ylog(p) \tag{1.3}$$

The y variable represents whether class k is the correct class given the current sample (y is 1 or 0), and the p variable represents the predicted probability of class k being the correct class. Cross-entropy loss is an example of a loss function commonly used in conjunction with categorical information, and thus highly utilized in classification problems. Cross-entropy loss is the cost function used throughout this work.

The progress made by deep learning has been astounding. Many modern libraries exist for implementing and training deep learning models. Pytorch and Tensorflow are leading examples of deep learning frameworks [86, 1], and are both capable of building large scale networks. Many applications have come online in recent years that were unrealized beforehand. Deep learning has been used to obtain state of the art accuracies on classification problems, speech recognition, robotic control systems, MRI anomaly detection, and segmentation, and object detection in general [39, 18, 60, 97, 62].

Recurrent versions of artificial neural networks (formally known as Recurrent Neural Networks (RNN)) have achieved state of the art on time series problems [126, 36], and modern deep learning architectures known as transformers use attention mechanisms to do amazing things with natural language processing [130]. However, all of these breakthroughs come at enormous cost. GPUs have begun to be incorporated

into supercomputers in vast quantities, as the capabilities afforded by deep learning comes typically with large amounts of computational expense. These expenses can be out of reach for individuals without access to such comprehensive supercomputing resources. Therefore, power savings are of integral importance in moving forward.

This dissertation focuses on ideas related to a different model of neural networks that have been shown to require immensely lower power: Spiking Neural Networks (SNN) [34]. In essence, an artificial neural network (ANN) as a phrase refers to the usage of non-spiking models. Spiking neural networks, on the other hand, refer to neural networks that propagate information with spikes. These are discussed further in the following section.

1.3 Spiking Neural Networks

Spiking neural networks (SNNs) are an increasingly popular model of computation [67, 34, 134]. Unlike their non-spiking counterparts, the neurons simulate spiking behavior similar to neuron and synapse behavior in real biological brains. A simple example of such a system is that of an integrate and fire neuron (I&F) [13]. Integrate and fire neurons accumulate electric charge until they reach a threshold. After the neurons cross the threshold, they send a pulse across their connected synapses. The synapses are generally weighted and can have individual delay values to simulate the length of synapses in the brain. Parameterization of these neuromorphic systems focuses on the neurons and synapses in many different ways, but all share this common behavior of spiking information propagation. Recurrent connections and weighted/delayed neurons incorporate real spatial and temporal information that most non-spiking neural network architectures fail to capture.

These sorts of models have only begun to be studied rigorously in recent years, and require different algorithms than traditional neural networks. The rise of spiking neuromorphic systems is the leading cause. Because of their theoretical capabilities [67, 113], they are able to tackle complex tasks such as spatiotemporal data processing [91]. A key question associated with SNNs is that of training,

both for general SNNs and for those developed with the intent to deploy on real neuromorphic systems. Specifically, determining the network's topology (number of neurons and synapses and their connectivity) as well as the parameters of the network (weights, delays, thresholds) is not a straightforward task. To achieve the full computational power of SNNs and thus of neuromorphic systems, we must develop effective algorithms for building SNNs tailored to real tasks.

In this work, we build on previous explorations of methods for training SNNs: training via evolutionary optimization, and reservoir computing. In particular, we explore network generation for use as the reservoir, using evolutionary optimization. We investigate the consequences of utilizing both of these training methods to produce networks for use on neuromorphic systems. Also, we analyze the general metrics associated with the training process for each approach. TENNLab architectures are used in conjunction to demonstrate the successful application of these training methods. The following sections provide a brief discussion of relevant TENNLab architectures.

1.3.1 NIDA

Neuroscience Inspired Dynamic Architecture (NIDA) is a 3d spiking neural network architecture [102]. The 3d aspect of this model allows for no constraints on connectivity, yet grounds the neurons explicitly with 3d coordinates. In other words, neurons exist at 3d coordinates, and synapses form interconnections between these neurons. 3d coordinate based neural networks are in contrast to many existing popular spiking neural network simulators. For example, Brian2 [120] and Loihi [21] allow for the formation of groups of neurons, not explicitly embedded in a topological space, which then may be connected arbitrarily with synapges.

NIDA utilizes a simple form of integrate and fire spiking neurons with refractory periods. The synapses are directed and have weights and delays. NIDA implements support for configurable long-term potentiation and long-term depression. It is a software-only architecture and thus does not have a hardware implementation. However, it is useful for comparison, as the low amount of structural constraints give

NIDA the ability to form complex representations. Catherine Schuman introduced the NIDA neural network model in her PhD dissertation, which serves as an excellent reference for more information [102].

1.3.2 DANNA2

DANNA2 stands for "Dynamic Adaptive Neural Network Arrays, version 2" [76]. It is a neuromorphic architecture that is is an iterative improvement upon DANNA, which was the first hardware architecture (FPGA & VLSI) developed within the TENNLab research group. DANNA2 is a hardware-based neural network, which means it is constrained to certain architectural properties. A software simulator exists, which allows for the training and testing of neural networks across various problems using different algorithms. It is also an integrate and fire model with configurable weights and delays. It implements LTP / LTD. The neurons also have a leak parameter. The topological embedding and constraints d differ greatly between DANNA2 and NIDA. The synapses and neurons are only placeable in certain arrangements because DANNA2 is a 2-dimensional model, whereas NIDA's three dimensions support highly variable configurations.

The FPGA and VLSI hardware constraints require network topologies and configurations that do not violate physical device reality. Therefore, the hardware imposes restrictions upon viable learning algorithms. It is necessary to follow these constraints to obtain the real-world application of the spiking neural networks. Therefore, studying such neuromorphic systems in both hardware and software simulation is integral to the advancement of spiking neural network research.

Because DANNA2 is a significant improvement upon the original hardware; the following dissertation experiments heavily use the DANNA2 system. However, we mention DANNA since it is the original architecture from which this one derives improvements.

1.3.3 mrDANNA

mrDANNA stands for "Memristive Dynamic Adaptive Neural Network Arrays." [15, 16]. It is a SNN architecture that has been developed to utilize memristors as an underlying nanotechnology. Full CMOS Memristors and Hafnium Oxide memristor implementations exist. The Hafnium Oxide implementation is a mixed-mode system designed for low-power operation, allowing for dynamic reconfiguration. The memristors are embedded between metal one and metal two layers in a custom post-fabrication integration at Suny Polytechnic Institute's Center for Semiconductor Research. The fabrication is CMOS, and the integration of memristive technology is highly unique in this regard. The average low resistance state and high resistance state are 10K and 150K Ohms. The mrDANNA simulator is implemented to represent this architecture and allow for the exploration and design of networks for this system using various algorithms. The mrDANNA architecture is not used in this dissertation but serves as a great example of a neuromorphic system that utilizes memristive devices to achieve low power. A reservoir computing system could utilize mrDANNA quite easily.

1.4 Evolutionary Optimization of Neuromorphic Systems

Genetic algorithms are a well-known category of algorithms used in optimization problems [79, 118]. They are useful when a direct analytical or numerical solution to an optimization problem is unavailable. They are capable of taking minimal feedback and still maintaining convergence properties. The basic idea derives from nature, analyzing, and combining/propagating populations through evolutions to derive and arrive at useful traits given certain problem constraints. In other words, genetic algorithms build directly on the concept of biological evolution. Each member (candidate solution) of a population contains properties that can be mutated and changed. Throughout generations, mutations and other changes known as crossover

(combining individuals of a population) are applied to the population in order to search a loss space for various solutions.

Evolutionary algorithms (a superset of genetic algorithms) have been utilized for many optimization problems. Based on the concept of natural evolution, these algorithms modify and adapt solutions throughout time in an attempt to discover successful combinations of their underlying properties. The TENNLab research group utilizes a sophisticated implementation of such an algorithm, called Evolutionary Optimization of Neuromorphic Systems (EONS) [108]. EONS has been applied to many problems across several significantly different architectures. We have used it to design networks we can apply to control systems (similar to OpenAI Gym) as well as non-linear classification problems. In EONS, the population of potential solutions contains many different spiking neural networks, each with varying topological properties and network parameterizations. The algorithm implements crossover and mutation, as well as various selection techniques. Provided a fitness function (a measure of success given a network and a well-defined problem), the algorithm will progress through epochs (generations) by changing members of the population in various ways to attempt to find improved solutions. The application of our implementation has been publicized in many different venues and has served as an excellent test bench for our group's affiliated neuromorphic hardware research [23, 108].

1.5 Reservoir Computing

Reservoir computing is a technique that uses either spiking or non-spiking neural networks as dynamical systems (called reservoirs) to filter and map information from one dimensionality to another [132]. Reservoirs with sufficient properties are capable of transforming complex non-linear data into a form that is linearly separable. Reservoir computing is similar to random projection methods found throughout statistics, data science, and various machine learning algorithms. The idea can also be approached from the standpoint of principle component analysis. As stated

above, the goal is to utilize neural networks as a feature map to translate from one form of dimensionality to another (generally higher) dimensionality in the hopes of transforming the data into a form that can be separated by simple linear regressions. This theory is useful for various problems. Typically, these problems are classification or regression-based.

The method is recent in scientific study, only dating back to 2001 [46, 69]. Thus, reservoir computing requires far more study before becoming useful in most practical applications. Nevertheless, it has shown much promise as a method of utilizing neural networks. Reservoir computing shares similarities with random projection methods. Reservoirs, in theory, do not require the direct training of reservoir weights (with caveats that will be discussed throughout).

There are two main versions of reservoir computing: Echo State Networks (ESNs) and Liquid State Machines (LSMs) [64]. Echo State Networks use traditional non-spiking recurrent artificial neural networks, whereas Liquid State Machines utilize spiking recurrent neural networks. Therefore, LSMs provide a nice case study of utilizing our group's neuromorphic architectures with a novel methodology in order to study non-traditional use cases. Further, with reservoir computing being such a new field of research, there are many research avenues the research that may be explored. This dissertation studies the use of LSMs with some of our group's models in novel experiments in order to attempt to unveil some better understanding for the TENNLab group, neuromorphic computing, and reservoir computing in general. We now describe the theory of liquid state machines in a more specific manner.

A Liquid State Machine (LSM) is a computational model incorporating the time-varying behavior of recurrent spiking neural networks as a filter for information. The LSM model includes three parts: an input layer, the liquid (i.e., the reservoir or recurrent spiking neural network), and a readout layer. In order to act successfully as the liquid or reservoir, a spiking neural network must have two fundamental properties: input separability and fading memory. Input separability means that the liquid filters the data in such a way that different states are reached provided different inputs. Without this property, a readout mechanism will be unable to differentiate

between the different inputs. For example, measures such as euclidean distance can be used to track rates of separability across output vectors. Fading memory is the requirement that a single input will not recursively propagate ad infinitum. One way to aid with this property is to introduce leaks to the architecture's neurons. Several measures of reservoir stability have been explored, such as the spectral radius and the Jacobian of the weight matrix [131]. Necessarily, these features can give numerical measures of reservoir stability with respect to a fading memory, and they have been used in studies of genetic algorithms for reservoir generation [27].

As mentioned above, to have theoretical guarantees of computation capabilities, a liquid state machine must have two properties: input separability and fading memory. If this is the case, it can be proven to be a universal function approximator via the Stone-Weierstrass theorem [68]. If a spiking neural network has these two characteristics, then it can act as the liquid for an LSM. Thus, the only training step for the spiking neural network component is ensuring that the network is sufficiently complex as to have both properties required of the liquid, and oftentimes these properties are inherent within recurrent spiking neural networks. This can make randomly generated liquids sufficient. However, in this work, we demonstrate that the likelihood of a randomly generated network working for a given problem depends on *a priori* information, such as a relative size for the randomly generated liquid. It also depends on the problem being examined. In this respect, EONS may help to generate successful topologies without needing to know these properties in advance. After the liquid has been stimulated to generate states for post-processing, the readout layer is trained to classify information from the provided state vectors. In particular, the readout layer is typically trained with linear regression. However, the definition of a Liquid State Machine is general enough to allow for different kinds of readout layers such as support vector machines, perceptrons, or multivariate logistic regression (softmax). In recent years, a new biologically inspired learning rule has been devised for teaching linear readout neurons to learn from local information that does not require intermediate representations of the spike trains for use by the readout neurons [125].

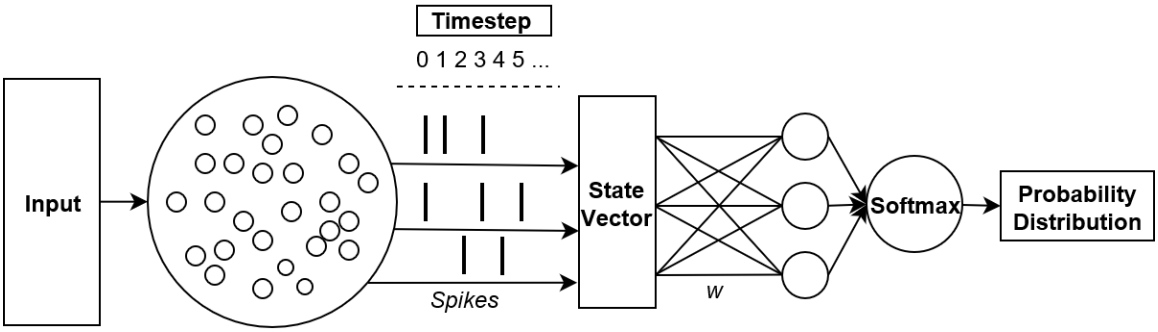


Figure 1.1: Diagram of reservoir computing. Input is passed into the reservoir (liquid in this case), then after a predetermined amount of time, the reservoir is sampled and the sample is passed into the readout layer as a feature vector. The readout layer is then trained on the features taken from the reservoir.

The software implementation of the above diagram (included in Appendix A) is modeled around utilizing any of the original hardware written within the framework as liquid state machines. In order to do this, we need to consider the theory of computation with LSMs. The input to our liquid can be represented as $u(.)$. The architectures are being utilized as the non-linear filter (liquid), and can thus be described with the function L^M. Lastly, our readout can be written as an arbitrary function, $f(x)$. Knowing this, we can model our reservoirs with the following mathematical equations:

$$x^M(t) = L^M(u(.))y(t) = f(x^M(t)) \tag{1.4}$$

The output of the liquid is time-dependent, making it equivalent to $x^M(t)$. The output from the readout layer is the composition of our readout function $f(x)$ and the time-variant liquid state $x^M(t)$. Thus, as demonstrated in the equation below, the readout function can be used to interpret the output from a liquid.

$$y(t) = (f \circ x^M)(t) = f(x^M(t)) \tag{1.5}$$

Here, we use a multinomial logistic regression (a softmax classifier) for the readout layer. The weight matrix is trained with gradient descent and the backpropagation

algorithm. Input is fed into the liquid, which transforms its state throughout the process of spiking events. It is this altered state (the outputs of the liquid) that may be captured for processing by the readout layer. After a predetermined number of simulation cycles, the state of the liquid is taken and passed to the softmax classifier. The state representation of the liquid is contained within a vector. The state vector Z that we utilize in this work consists of the number of spikes an output produces throughout the simulation and contains elements equal to the number of outputs of the network. This is particularly easy to compute. A state corresponding to a particular input may be reused; it does not have to be re-simulated for every training step. Mathematically, the representation is described as follows:

$$Z_i = \sum_{j}^{N} Output_{i,j} \tag{1.6}$$

N is equivalent to the number of simulated time steps. The vector $Output_{i,j}$ contains a 0 if the output neuron i did not fire on timestep j, and a 1 if it did fire. Z_i represents the state for output i. The state vector is computed after simulation and passed to the readout layer. Instead of a snapshot of the entire state of the liquid at a particular time, the state vector tracks the dynamical behavior of the network through output spikes. This can be likened to classifying using only the waves generated within a liquid instead of the state of the entire body of liquid. Note that this is merely one possible state representation used for the ease of computation. Other possibilities exist and can improve classification performance through the use of more extensive information from the spike trains, such as low-pass exponential filtering of the generated spike signals.

The representation described above is passed through the readout layer to determine the final output. The readout layer is a multinomial logistic regression, which is essentially a perceptron (linear combination of weights demonstrated in a previous equation) that gets passed through the softmax function to map to a probability distribution. The softmax function can be seen in the following equation:

$$\sigma z_i = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}} \tag{1.7}$$

During the training phase, the readout layer then computes the error gradient of the softmax and squared error cost function. Lastly, backpropagation is used to update the weights.

The counting kernel is demonstrated with an experimentally generated liquid on the classic iris dataset [28] in figure 1.2. Three samples were filtered through the liquid; one of each class. Interestingly, the individual classes all show explicit ranges of numbers of fires.

1.6 TENNLab Software Structure

The TENNLab Neuromorphic research group is multi-faceted [90, 88, 108, 23]. Several neuromorphic architectures have been designed, and evolutionary algorithms have been applied to train spiking neural networks to perform complicated tasks on topologies specific to the hardware models. In order to train on and test these models, simulators were developed in-house and utilized alongside various algorithms across a suite of applications. The structure of the TENNLab software framework supported the development of these simulators and applications, as well as the usage of training algorithms upon them. We briefly describe the structure of the software framework within which this work resides.

The software structure exists around three core components: architectures, algorithms, and applications. The architectures are composed of the previously described models and provide both software simulators and hardware hooks for dealing directly with the neuromorphic hardware. The algorithms component consists primarily of EONS. In the applications component, we maintain a variety of control problems as well as classification systems for dealing with typical machine learning problems as well as utilizing our own neuromorphic applications test benchmarks.

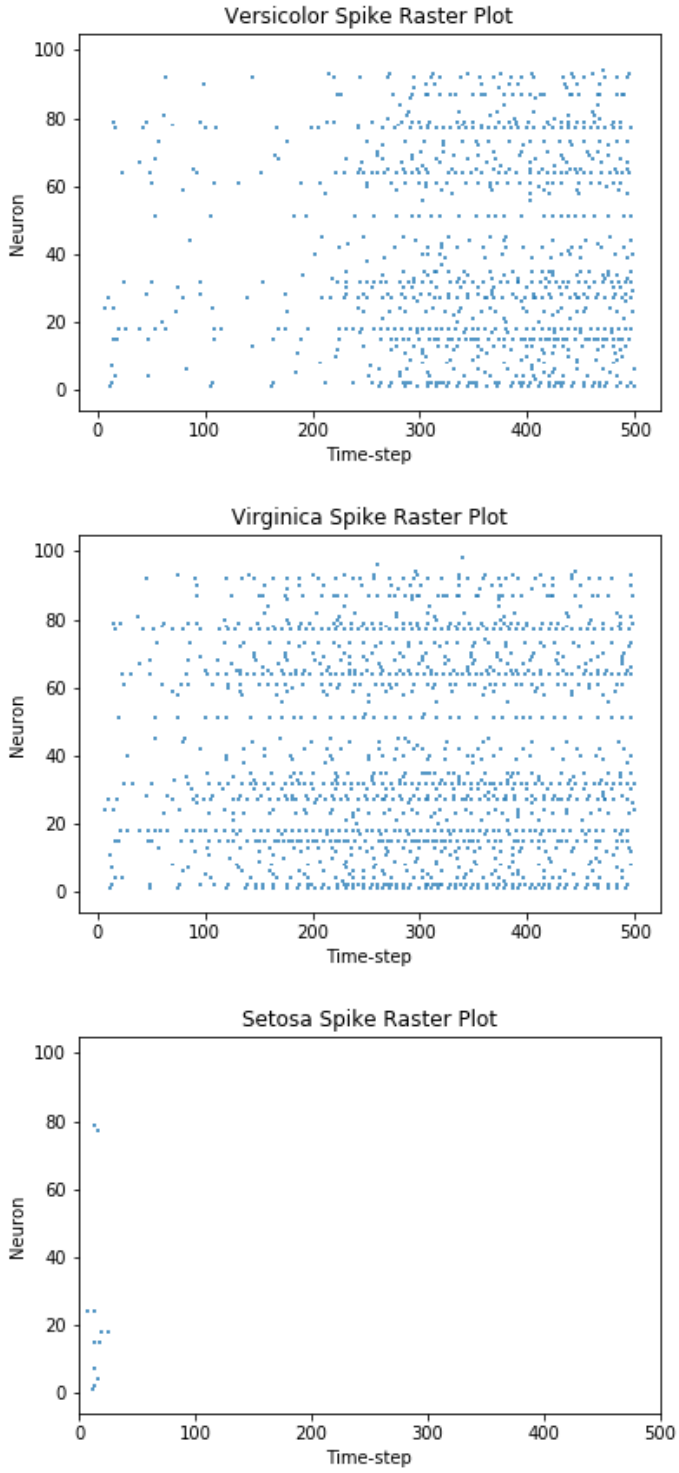


Figure 1.2: This is an example of how the counting kernel causes separation in a reservoir. The classes are the three categories found in the well-known Fisher's Iris dataset [28]. The number of fires is clearly distinguishable between the three examples.

Our architecture stack includes systems components for connecting directly to FPGA hardware via USB. We have the capability to train neural networks on these neuromorphic systems "in the loop", e.g., loading onto hardware, simulating, then unloading networks. Our software stack also provides a translation component for translating input data vectors into spike trains [107] for easy conversion of arbitrary application data to neuromorphic systems. Further, our software stack allows for the creation and handling of multiple neuromorphic simulators at once in order to evaluate EONS fitnesses in a multithreaded environment. This allows for massive speedup over the linear processing of neural network data. In this work, both the NIDA and DANNA2 architectures are used in this fashion across a variety of problems.

Our application component contains multiple applications that are capable of interfacing with any of our neuromorphic architectures. It is effortless to swap out architectures. Primarily, classification problems are explored in this paper as a measure of success between EONS and reservoir computing. However, in the final experimental chapter, this dissertation delves into reinforcement learning in combination with reservoir computing in order to attempt to evaluate the capability of neuromorphic reservoirs aiding in the translation of control systems data for more natural learning and traversing of the loss space.

1.7 Dissertation Structure

The remaining chapters of this dissertation are structured in a natural flow to allow readers the opportunity to understand where reservoir computing fits into the overall neuromorphic structure. Chapter two provides a brief literature review of relevant information. Chapter three incorporates popular deep learning models as well as pure EONS generated networks. These models are compared with grid search generated reservoirs, thus providing insight into the reasonable application of reservoir computing when contrasted versus different methodologies. Chapter four explores utilizing the TENNLab in-house genetic algorithm, EONS, to directly generate successful liquid state machines. Chapter five documents an exploration of

combining reinforcement learning and liquid state machines on LIDAR based control applications. Finally, the dissertation concludes with a discussion on informative results and future directions for any further studies. The appendix includes the exact C++ implementation of the readout layer that was used in this work.

Chapter 2

Related Work

2.1 Overview

This dissertation depends on the combination of spiking neural networks and non-spiking neural networks. The spiking neural networks serve as liquid state machines, and the non-spiking neural networks are simple multinomial logistic regressions used as a readout layers. This chapter will provide a brief overview of relevant information. Many recent publications discuss useful facets of these topics, and thus serve as important references for the basis of this work.

2.2 Spiking Recurrent Neural Networks

Spiking neural networks gained prominence in neural network literature in the late 1990's. One of the most important distinguishing factors between spiking neural networks and previous generations is the non-trivialization of time in the computation of the network; that is, the timing of events in a spiking neural network is integral to the way that computation occurs [67]. Since then, they have become increasingly popular, partially due to the rise of spiking neuromorphic hardware [109]. Spiking recurrent neural networks (SRNNs) are those that allow less restricted or unrestricted

connectivity between neurons in the network; that is, the networks are not organized in a layered structure that is typical with other neural network types.

As noted in the introduction, a key issue associated with spiking neural networks in general, and spiking recurrent neural networks in particular, is how to train them. Gradient-descent or back-propagation-based spiking neural network training algorithms have been proposed [10, 112], but they typically rely on a pre-determined network topology and may not fully leverage the full computational capabilities of the networks. One of the most common training approaches for spiking neural networks in general (including spiking recurrent neural networks) is spike-timing dependent plasticity or STDP [72]. Again, in this case, properties of the networks, like delays, may not be defined by the training algorithm, leaving it to the user to define, often in an ad hoc, trial-and-error way.

Defining network topology in a systematic way has been done for both traditional neural networks and spiking neural networks using genetic algorithms or evolutionary approaches in the field of neuroevolution [30]. Neuroevolution methods have been used to determine all aspects of neural networks, including determining network topology and parameters simultaneously [119] and determining network topology while utilizing a more traditional weight training algorithm [3]. Neuroevolution methods have also been used specifically for evolving spiking neural networks [52, 87] and spiking neural networks for neuromorphic implementation [108, 14].

Neuromorphic hardware implementations are driving much of the work in this field. Many high-profile implementations have been released to the general public in the last decade. Namely, IBM's TrueNorth and Intel's Loihi are two of the most prominent neuromorphic architectures to date [2, 21]. Other well-known projects include SpiNNaker, the Human Brain Project, and Neurogrid [93, 71, 9]. A survey paper was published in 2017 detailing important works in neuromorphic computing/spiking neural networks, and thus serves as an excellent reference for more information [110].

2.3 Reservoir Computing

Reservoir computing is an approach for utilizing recurrent neural networks to filter information with the hope of finding useful and interesting data representations [65]. The two main reservoir computing approaches are echo state networks and liquid state machines. Echo state networks [46] use non-spiking recurrent neural networks for the reservoir. Liquid state machines [69] use spiking recurrent neural networks as filters for non-linear input separability. Reservoir computing and liquid state machines have successfully utilized spiking recurrent neural networks for a variety of applications, such as robotics controls, object tracking, motion prediction, pattern classification, signal processing, and time-series prediction/classification [4, 11, 47].

The two main subgroups, as mentioned previously, are Liquid State Machines and Echo State Networks. It is important to provide helpful references to relevant literature broken down by specific technique. Liquid State Machines and Echo State Networks are fundamentally different applications of the same principle, and therefore require individual subsections.

2.3.1 Liquid State Machines

Liquid State Machine (LSM) literature discusses many different angles of the technique. When compared to other branches of research, however, publications on the topic of LSMs are seemingly sparse. This is a major factor in deciding to focus dissertation research on the subject. Yet, there are publications demonstrating success with liquid state machines across a variety of problems. Word recognition, movement prediction, pattern recognition, speech recognition, and time-varying signal classification all serve as excellent demonstrations of LSM successes [133, 12, 140, 101, 17].

Using memristive devices and neuromorphic architectures for liquid state machines is an important driver of active research [57], [66], [116], [55, 25]. The ability for spiking architectures to provide interesting complex computation without training

the full network leads to energy efficient hardware applications. Further, reservoir computing can be implemented with other sufficiently dynamical physical systems. For instance, research groups have demonstrated implementations of reservoir computing using actual liquid [26], optoelectronics [85], and quantum systems [31].

Much of the liquid state machine literature focuses on hardware design. The theory of liquid state machines makes it particularly useful for experimental machines, considering that the weights of the spiking network don't have to be explicitly trained. More references for relevant liquid state machine work are provided in the related chapters, specifically because it makes it easier to interpret the included work. A greater number of successful applications have been demonstrated with echo state networks than liquid state machines, perhaps due to the availability of known linear solutions for traditional recurrent neural networks. The following section further discusses Echo State Networks.

2.3.2 Echo State Networks

Reservoir computing can also be performed with non-spiking neurons. The non-spiking artificial neural network variant of reservoir computing is known as Echo State Networks [46]. A network of recurrent interconnected non-linear non-spiking neurons can be stimulated with inputs. Then, the state can be collected after a pre-determined number of time-steps. A readout function is applied to map from the domain of the reservoir state into that of the selected problem. Echo state networks have been utilized in many different problem domains, and are reported to have success in a wide variety of situations [66, 64, 63] such as time-independent classification [4], human activity recognition with wireless sensor data[84], pattern generation [48] (for example).

Echo State Networks being continuous allows for many different implementation decisions. Specifically, one may change the type of activation function being used amongst traditional artificial neural network choices (tanh, ReLu, Sigmoid, etc.). Further, the type of recurrent neuron may be changed (LSTM, GRU, etc.). ESNs can be implemented using any typical and popular neural network framework; e.g.

Tensorflow, PyTorch, Keras, CNTK, Theano, Caffe, and more [86, 1, 19, 111, 127, 50]. Being the non-spiking variant, ESNs do not have the same constraints as their spiking counterparts. The network connections are directly differentiable and can be represented with linear systems, meaning that one can solve directly and change the weights to optimize certain network properties. Many papers and works exist studying the topic of Echo State Networks [65]. However, this dissertation work relies on spiking neural networks, and therefore Liquid State Machines, which do not have the same theoretical underpinnings as Echo State Networks [69].

2.4 Genetic Algorithms

Many bodies of research have spent extensive time studying genetic algorithms and evolutionary algorithms [54, 138, 143, 129]. In particular, there are many works specifically surrounding applying genetic algorithm techniques and principles to neural networks outside of the TENNLab EONS implementation [6, 33, 37]. Notably, the NEAT algorithm is a prominent genetic algorithm within the neural network community [119]. Oak Ridge National Laboratories has a group dedicated to applying evolutionary principles to Deep Learning topology and hyperparameter searches with a technique titled MENNDL [141]. These techniques have all seen success, and draw extensively from evolutionary and genetic algorithm principles. It is from this vein of research that the EONS algorithm was derived and applied to spiking neuromorphic architectures. Evolutionary Optimization of Neuromorphic Systems (EONS) implements the genetic algorithm operations of crossover and mutation, population propagation, and individual selection for epoch progression. EONS is described heavily in many different publications, which can be referred to for further information [23, 108, 106].

2.5 Reinforcement Learning

Reinforcement learning is a fundamental machine learning paradigm [123, 51]. It relies on the experience an agent gains in an environment to reinforce and update that agent's behavior. The reinforcement paradigm of machine learning can be better understood in consideration against supervised learning, wherein the desired answer is known and provided to the learning system. There are many different algorithms for Reinforcement Learning: q-learning, policy gradient, and actor-critic algorithms just to name a few [137, 124, 53, 123]. Reinforcement learning is usually described as a markov decision process. In recent years, reinforcement learning has seen major strides in the field of deep learning [81]. Amazing feats have been demonstrated in largely complex space and applications from AlphaGo and AlphaStar [115], which play Go and Starcraft respectively, to Deep Q-Networks for Atari games in the OpenAI gym environment [80, 128]. These works all make use of recent advances in deep learning techniques to optimize reinforcement learning objective functions with gradient descent and backpropagation. The theoretical underpinnings of reinforcement learning are well defined, understood, and documented. Much of the theory exists within the realm of dynamic programming and attempting to iteratively improve upon the decision making process (policy) that an agent uses within its environment. A few works have explored doing reservoir computing and reinforcement learning with varied success. In particular, echo state networks have been used to learn robotic controls [7], and liquid state machines have recently (towards the end of writing this dissertation) been used in conjunction with q-learning to play atari games while claiming mixed results [92]. A further treatment on the subject is included in chapter five, which details an exploration of the combination of TENNLab liquid state machines and reinforcement learning.

Chapter 3

A Comparison of Neuromorphic Classification Tasks

3.1 Overview

This chapter is reproduced work from a publication written by the thesis author [96].

A variety of neural network models and machine learning techniques have arisen over the past decade, and their successes with image classification have been stunning. With other classification tasks, selecting and configuring a neural network solution is not straightforward. In this chapter, we evaluate and compare a variety of neural network models, trained by a variety of machine learning techniques, on a variety of classification tasks. While Deep Learning typically exhibits the best classification accuracy, we note the promise of Reservoir Computing, and evolutionary optimization on spiking neural networks. In many cases, these technologies perform as well as, or better than Deep Learning, and the resulting networks are much smaller than their Deep Learning counterparts.

Neural networks and machine learning are ubiquitous in today's computing landscape. Given the incredible successes of Deep Learning approaches to image classification, any scientist or researcher, when confronted with a task that involves the classification of data, is compelled to try some sort of neuromorphic machine

learning approach to solve his or her problem. The researcher quickly discovers, however, that applying the successes of current neuromorphic approaches to problems beyond image classification is challenging, and there is very little credible guidance.

Besides Deep Learning, there are other neuromorphic approaches to data classification. Several methods rely on highly recurrent neural networks, rather than the feed-forward neural networks employed by Deep Learning, to attack the problem. Two notable examples are *Reservoir Computing (RC)*, where a fixed and highly recurrent "reservoir" is employed as a pre-processing step to convert complex data into data that is linearly separable, and *Evolutionary Optimization of Neuromorphic Systems (EONS)*, where highly recurrent spiking neuromorphic systems are trained via genetic algorithms to solve specific problems.

In this section, we embody a researcher who desires to use neuromorphic machine learning to solve some data classification problems, and we evaluate six approaches that such a researcher might consider. Four of them are Deep Learning variants, available in the Keras Deep Learning framework. The other two employ RC and EONS to leverage a spiking neuromorphic system called NIDA, to classify data. In each approach, we do *not* attempt to refine the approach to each specific problem, as we are embodying a scientific researcher who desires to use neuromorphic machine learning as a tool, rather than a computer scientist researching neuromorphic computing. This is important, as it allows us to evaluate our systems without fine-tuning due to domain specific knowledge of neuromorphic computing. We found that, in general, there was not one machine learning approach that performed best overall. However, both the spiking neural network (RC and EONS) approaches resulted in much smaller network sizes than the Deep Learning approaches, and thus can result in more size and power efficient neuromorphic implementations.

3.2 Neural Networks and Machine Learning Algorithms

The general approach of neuromorphic machine learning is similar across all of the techniques that we evaluate in this paper. We are given a collection of data points that are partitioned into different classes. We use this data to train the components of a neuromorphic system (e.g., the weights of the synapses), so that the system may be employed to predict the classes of subsequent data. The components that get trained, and the techniques for training, differ among the systems that we evaluate. We explain these in the following subsections.

3.2.1 Deep Learning Approaches

Deep Learning is a term that encompasses layered, (mostly) feed-forward neural networks that are trained with back-propagation. At a high level, the neurons in the system hold values, and synapses propagate values from one neuron to another. Synapses have weights, and each neuron's value is calculated as a function of its incoming synapses' weights and propagated values. The neurons are partitioned into layers, and the connectivity from layer to layer is predefined. *Dense* layers have full connectivity from neurons in the previous layers, while other layers such as *convolutional* or *maxpooling* layers have more restricted connectivity.

A *model* in a Deep Learning system is the definition of the layers, their connectivity, and the mathematical functions that compute neuron values from their incoming synapses. Once a model is defined, the training process uses back-propagation to define the weights of the synapses. Back-propagation uses a gradient descent-style process to iteratively reduce the error of the system when applied to the training data. If a model is more complex, the training process will be slower; as such, many researchers make their models, and often the training weights, public so that other researchers may benefit from their training process.

There are a variety of open-source Deep Learning systems such as TensorFlow, CNTK, and Theano [1, 111, 127]. The Keras Deep Learning framework is a high-level framework and API in Python, that allows one to design and employ Deep Learning models with any of these as their back-end [19]. We have developed a very simple data classification framework in Keras that allows us to employ multiple models to classify data, and in this paper, we evaluate four models:

- **Perceptron** is a simple model with just two layers of neurons: the inputs and the outputs. They are fully connected by synapses. As such, this model can perform linear separation of its data. The output is passed through a softmax activation function to generate a probability distribution for classification purposes.

- **Multi-Layer Perceptron** is composed of three fully connected layers, each containing 64 neurons, employing the ReLU activation function, with a softmax at the end. Sometimes termed "vanilla neural networks," Multi-Layer Perceptron networks are known to be good classifiers, with the multiple layers and non-linear activation functions allowing them richer functionality than linear classifiers [38]

- **Conv**: "LeNet-5" is a well-known convolutional model first developed for handwriting recognition [59]. It is easily modifiable for different input structures, and therefore is our archetypical model for convolutional Deep Learning networks.

- **LSTM**: "Long Short-Term Memory" is an enhancement to the neuron structure of standard feed-forward neural networks to add some recurrence, and ideally some memory, to the networks [43]. Stacking LSTM layers is a commonly used technique in order to better learn higher dimensional features and time-series data [35]. Our LSTM implementation contains three layers, each with 32 cells.

3.2.2 Spiking Recurrent Neural Networks

Spiking neural networks work very differently from the Deep Learning neural networks described above. In spiking neural networks, there is an explicit time component, and rather than holding values, neurons hold *charge*. Synapses transmit charge over time, and when a synapse's charge arrives at a neuron, it is added to the neuron's charge value. If this neuron's charge exceeds a threshold, the neuron *fires*, resetting its charge value to some base value and sending charge out along its outgoing synapses.

In a spiking neural network, neuron thresholds, synapse weights and synapse delays are configurable. When the networks are recurrent, there is no constrained structure as there is with Deep Learning networks. The goal of training a spiking recurrent neural network is to define the connections, thresholds, weights and delays so that the network can "solve" an application. With a classification application, the values of the data points must be converted into spikes, and output spikes must be converted into classifications. We will describe how that works in Section 3.2.3 below.

The spiking neural network system that we use is called Neuroscience Inspired Dynamic Architecture (NIDA) [104]. This is a very simple system featuring neurons configured in three dimensional space and unlimited connectivity between neurons. Weights and thresholds are floating point values, and delays correspond to the Eudlidean distance between neurons. NIDA has a few additional features such as refractory periods for neurons, where they may accumulate charge without firing, and long term potentiation/depression, (LTD/LTP), where synapse weights grow and shrink according to how responsible they are to their target neuron's firing. NIDA has demonstrated success at applications such as data classification and control [105, 108, 103].

3.2.3 EONS: Evolutionary Optimization of Neuromorphic Systems

Spiking recurrent neural networks like NIDA are not static and tightly constrained as feed-forward neural networks are. Thus, they cannot be easily programmed by back-propagation, and other techniques must be employed to make them work effectively. EONS [108] takes a genetic algorithm approach to training networks. In particular, the following steps are taken in EONS to generate a spiking recurrent neural network for a particular application:

1. An initial population of networks is generated randomly. Heuristics may be employed to intelligently initialize networks, for example, to force input neurons to have paths to output neurons.

2. Each network in the population is evaluated and given a fitness value. This is done by having the application apply a training suite of tasks to it (e.g., sweeping through a training set of data) and measure its success (e.g., calculating the accuracy of the classification).

3. The members of the population with the highest fitness are selected for reproduction, which involves mutating parameters of single networks, and performing crossover operations on pairs of networks.

4. Return to step two with the population composed of these newly generated networks.

5. Quit when the fitness achieves a desired threshold, or after a specified period of time has passed.

EONS is advantageous because of its generality — so long as an application can define its fitness suite and a neuromorphic system can define the reproduction operations, then EONS can train networks. Its major problem is that its search space can be colossal, and therefore it can be slow (or unable) to converge. EONS has

been applied successfully to various control and classification problems on multiple neuromorphic systems [106, 108, 77, 89].

For EONS, we use the TENN-Lab Hardware/Software Co-Design Framework [89], which provides application and device support for spiking neuromorphic systems. This framework includes a general classification application, which trains networks for given labeled classification data sets. The application assumes that the each data point is composed of $(r \times c)$ values. It then employs a network with r input neurons and inputs the values in c groups, one per neuron. The groups are separated by 5 units of simulated time. The values themselves are converted into spikes normalized from -1 to 1.

There is one output neuron per class. For each data set, the network runs for N time steps from when the first wave of values begins, N being chosen based on the application. At the end of the N time steps, the output neuron that fires the most determines the classification. The TENN-Lab framework supports multiple neuromorphic computing architectures, one of which is NIDA.

3.2.4 Reservoir Computing

Reservoir Computing [45, 70] is a paradigm that attempts to harness the power of a highly recurrent neural network ("the reservoir") by training a "readout" layer to interpret its output. A high level diagram is in Figure 1.1. In this work, the reservoir is a large NIDA network, generated randomly. It processes its data for classification like the EONS application, with r input neurons for $(r \times c)$ input values. Each network has 100 output neurons. During the spiking neural network simulation, firings are counted for the last M time steps, with M being chosen based on the application. These counts compose the state vector of Figure 1.1, which is then input to the perceptron layer. The outputs of the perceptron layer are then passed to a softmax layer, as in the Deep Learning network. The reservoir in this system does not change once it is created. For NIDA, this means that the LTD/LTP synaptic plasticity process is turned off. Only the weights of the perceptron layer are trained using back-propagation.

Most research on RC is heavily laden with mathematics. One body of work employs non-spiking networks for the reservoir. These are called Echo State Networks [45]. A second body of work employs recurrent, spiking networks like NIDA. These are called Liquid State Machines [70]. A third body of work employs single neurons and delayed feeback loops as their reservoirs. These are called Time Delay Reservoirs [8]. With all three models, "good" reservoirs are defined by properties such as, for example, *input separability* and *fading memory* for Liquid State Machines [70]. Many works explore optimizing based on these parameters [27], but that expands beyond the thesis of this paper.

3.2.5 Summary of Neural Networks and Machine Learning Algorithms

Table 3.1 summarizes the neural networks and machine learning algorithms that we employ in this paper. We note that while Keras and Tensor Flow are both open-source software packages, at present the TENN-Lab software framework is not. The TENN-Lab authors welcome collaborations with application and neuromorphic architecture teams who wish to explore their software.

Table 3.1: Summary of the neural networks and machine learning algorithms employed in this paper.

Name	Software	Neural Network	Training
Perceptron	TENN-Lab	Feed Forward	Back-propagation
MLP	Keras / TensorFlow	Feed Forward	Back-propagation
Conv	Keras / TensorFlow	Feed Forward	Back-propagation
LSTM	Keras / TensorFlow	Nodes with recurrency	Back-propagation
EONS	TENN-Lab / NIDA	Spiking, Recurrent	Evolutionary Optimization
Reservoir	TENN-Lab / NIDA	Spiking, Recurrent + Perceptron	Random + Back-Propagation

3.3 Classification Applications

This section introduces and briefly describes the classification data sets that we explore in this work. The data sets come from a variety of sources and have different properties that make them interesting.

3.3.1 Iris, Breast Cancer, Pima

These are three well-known static classification datasets, available from the UCI Machine Learning Repository [61]. By static, we mean that there is no temporal component to this data. For EONS this means that c equals one. This simplification required us to change a few of the neural networks models. For **Convolutional**, we simplify the LeNet-5 model from a 2D convolutional model to a 1D model. For **RC**, because Reservoir Computing relies on dynamics generated within the reservoir, the features were mapped into multiple pulses over time for continuous stimulation. This is necessary for high accuracy classification of static tasks in reservoir computing [4].

The **Iris** data set is quite old [29] and is very frequently used as a first classification task for machine learning experiments. It consists of 50 data points for each of three classes of flowers: Setosa, Virginica, and Versicolor. Each data point has four values: the length and width of both the sepals and the petals. The Wisconsin Breast Cancer **(WBC)** dataset is composed of data collected from digitized images of a fine needle aspirate of a breast mass [121]. There are 699 data points, composed of 10 data values per data point. The classifications are whether the mass is benign or malignant. The **Pima** Indians Diabetes dataset is a set of 768 multivariate samples from patients that either have diabetes or do not. The goal is to determine whether a given patient exhibits the symptoms based on the number of pregnancies, glucose concentration, blood pressure, skin thickness, insulin levels, body mass index, pedigree function, and age.

3.3.2 Satellite Radio

The **Radio** dataset is published by DeepSig, Inc. [24], and contains 128-sample snippets of complex-valued temporal radio signals that have been modulated according to 11 different waveforms and 20 different signal-to-noise ratios. Each waveform contains the same number of snippets per signal-to-noise ratio, and there are between 4,200 and 24,940 snippets per waveform. The curators of this data have applied a custom 5-level Deep Learning network for classifying this data, and achieved roughly 95% accuracy when focusing on data with the highest signal-to-noise ratio [83]. For our testing, we selected one modulation type (8PSK) and tested classification performance of that type vs. all of the others, with the highest signal to noise ratio. We trained on 667 snippets, equally divided between 8PSK and the others, and tested on 166 different snippets, also equally divided. We chose the 8PSK modulation type due to its misclassification with other signals in [83], making it good for individual comparison. We show two example snippets from this data set in Figure 3.1.

3.3.3 Electroencephalogram (EEG)

The Electroencephalogram (EEG) is a measure of activation signals in the brain over time. An EEG can contain lots of interesting information within the signal. Because of this, statistical measures have been used to extract relevant and useful metrics.

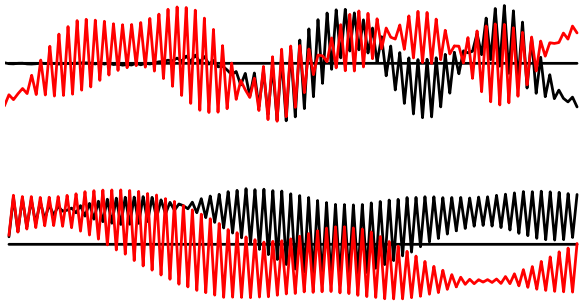


Figure 3.1: Two example snippets from the **Radio** dataset: 8PSK on the top, and GFSK on the bottom. Real values are plotted in black, and imaginary values in red.

Many works explore onset detection of epilepsy from EEG signals [116]. However, in this work, we perform standard classification rather than onset detection. The dataset used for classification is made publicly available by Andrzejak [5].

The dataset has five sets of EEG signals labeled A through E, composed of 23.6 seconds of signals recorded with a sampling rate of 173.61 Hz, for a total of 4,097 samples per data set. For classification, we employed 200 sets of signals split equally between healthy (from set A) and epileptic (from set E). We display two of these data sets in Figure 3.2. To be consistent with the literature, we used 80 signals from each set for training, and 20 for testing [122].

3.3.4 Consonants vs. Vowels

The **TIMIT** dataset contains audio files of ten sentences spoken by each of 630 speakers, sampled at 16 kHz. We convert each WAV file into Mel-frequency Cepstral Coefficients (MFCCs): vectors of 12 coefficients, where each vector represents a slice of the spoken sentence [22]. The MFCCs are computed every 10 ms, over a sample size of 25 ms. To the 12 coefficients, we add a 13th value, the log energy [142]. An example phoneme spectrogram is included in Figure 3.3. Being able to discern between vowels and consonants has important implications for speech recognition [20]. As such, we have partitioned the data set into two classifications: vowels and consonants. To be consistent with the work by Norton and Ventura [82], we employed a training set of

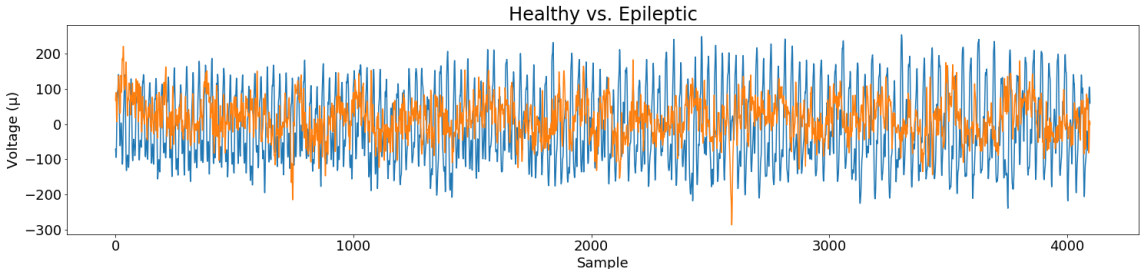


Figure 3.2: Example time series plots of two EEG signals: healthy (orange), and epileptic(blue).

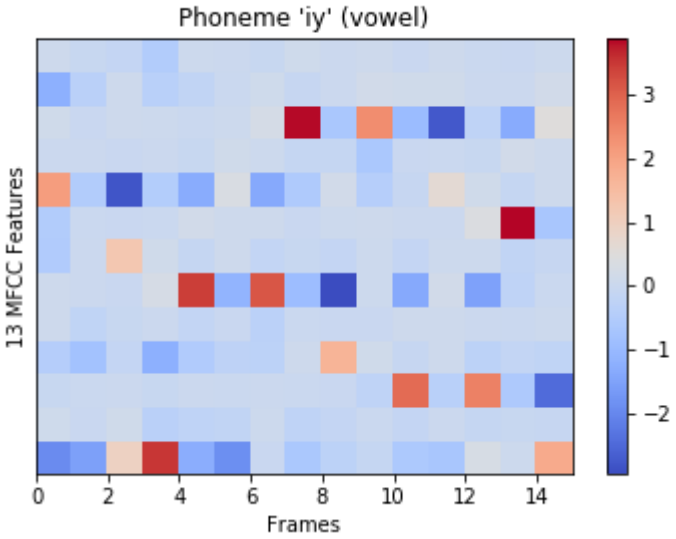


Figure 3.3: Spectrogram of MFCC features from a spoken phoneme 'iy'; a vowel. Each column is a MFCC vector with 13 values, calculated every 10 ms over 25 ms of samples.

2000 values, equally split between consonants and vowels, and a testing set of 200 values.

3.3.5 Summary of Data Sets

Table 3.2 summarizes the data sets that we explore in our experiments. For each data set, we summarize the $(r \times c)$ values for each data point that is input to the neural networks, the number of classifications, the partitioning of data points into training and testing, the number of time steps for the spiking neuromorphic networks, and the output window over which spikes are counted for the RC test.

3.4 Experimental Setup

The Deep Learning tests were executed on an Intel Core i7-7700 CPU, running at 3.60 GHz with 16 GB of RAM. Because EONS and RC are more computationally demanding during training, we performed them on a 44-node, 1772-core cluster of

Table 3.2: Summary of classification data sets and their parameters as they affect the various machine learning algorithms. N, which corresponds approximately to spiking neural network run-time, applies only to EONS and RC, and M, which corresponds to when output pulses are counted, applies only to RC.

Name	r	c	Classifications	Training Data Points	Testing Data Points	N (Time Steps)	M (Output Window)
Iris	4	1	3	75	75	440	330
WBC	9	1	2	560	139	440	330
Pima	8	1	2	615	153	440	330
Radio	2	128	2	667	166	1400	1200
EEG	1	4097	2	160	40	4100	4000
TIMIT	13	48	2	2000	200	500	450

AMD Opteron CPUs running at 2.3 GHz. Eight of the nodes have 24 GB of RAM, and the remaining nodes have 96 GB of RAM.

With the Deep Learning models, the $(r \times c)$ data points were flattened and applied into the networks simultaneously; with EONS and RC, they were input in c waves of r points, as described in Section 3.2.3. For Deep Learning, we ran 10 training runs of each method, each with a different random number seed, and trained for 1024 epochs. For EONS, we ran 100 training runs, each with a population size of 100 networks, for one hour each on a single CPU core. For RC, we generated 100 random reservoirs, and then trained the readout layer for 1024 epochs.

3.5 Results

The classification accuracies on the testing data of the best network for each model are shown in Table 3.3 and Figure 3.4. The best classification accuracy for each data set is on par with, or exceeds previously published data [108, 83, 82, 122].

Perhaps the most important thing to note from the results is that there is no consistent "best method" over all of the data. EONS displays the best classification accuracy for **Iris**, **WBC** and **EEG** (tied with **Conv**); the Convolutional Deep Learning method displays the best accuracy for **Radio**, **EEG** and **TIMIT**; and **LSTM** and **Reservoir** tie for the best at **Pima**.

Table 3.3: Percentage accuracies of each model on each data set. Each number presents the best classification accuracy on the testing set of data.

Model / Data	**Iris**	**WBC**	**Pima**	**Radio**	**EEG**	**TIMIT**
Perceptron	97.38	48.40	67.54	50.70	54.05	77.57
MLP	95.33	94.71	68.37	80.53	48.25	83.73
Conv	96.13	96.45	67.70	93.95	99.00	85.20
LSTM	96.67	95.71	79.22	83.64	45.00	83.40
EONS	98.66	99.28	78.57	71.00	99.00	83.00
Reservoir	93.33	96.47	79.22	73.00	98.00	85.00

With the exception of **Iris**, on which all of the models perform well, the **Perceptron** model classifies significantly worse than the other models. This is unsurprising, since the **Perceptron** model has no hidden layer, and thus restricted computational abilities. The Multi-Layer Perceptron (**MLP**) improves accuracy sigificantly over the single Perceptron with the **WBC**, **Pima** and **TIMIT** data sets. The **Conv** model performs poorly only on the **Pima** data set, and it performs better than all of the other models on the **Radio** data set. As described by O'Shea *et al*, this is because the **Radio** data set exhibits features similar to image classification [83]. It is surprising that **LSTM** performs poorly on the **Radio** and **EEG** data sets, given that they are time-series data sets, for which **LSTM** was developed. We surmise that this is due to two separate reasons. For EEG, the number of samples (4097) is still considered to be a very long sequence, and the LSTM paradigm is impacted by the vanishing gradient problem. By exploring parameter settings in Keras and breaking the sequence into subsequences, **LSTM** could improve. Since a goal of this paper was to inhabit the perspective of a scientist wanting to classify data, and not a computer scientist exploring Deep Learning, we intentionally did not spend time exploring parameter settings.

The spiking neuromorphic models perform well with one exception. The exception is on the **Radio** data set, which obviously has features that the two training methods could not discover. We intend to explore this data set more thoroughly to discover why the two models perform so much worse than the Deep Learning models.

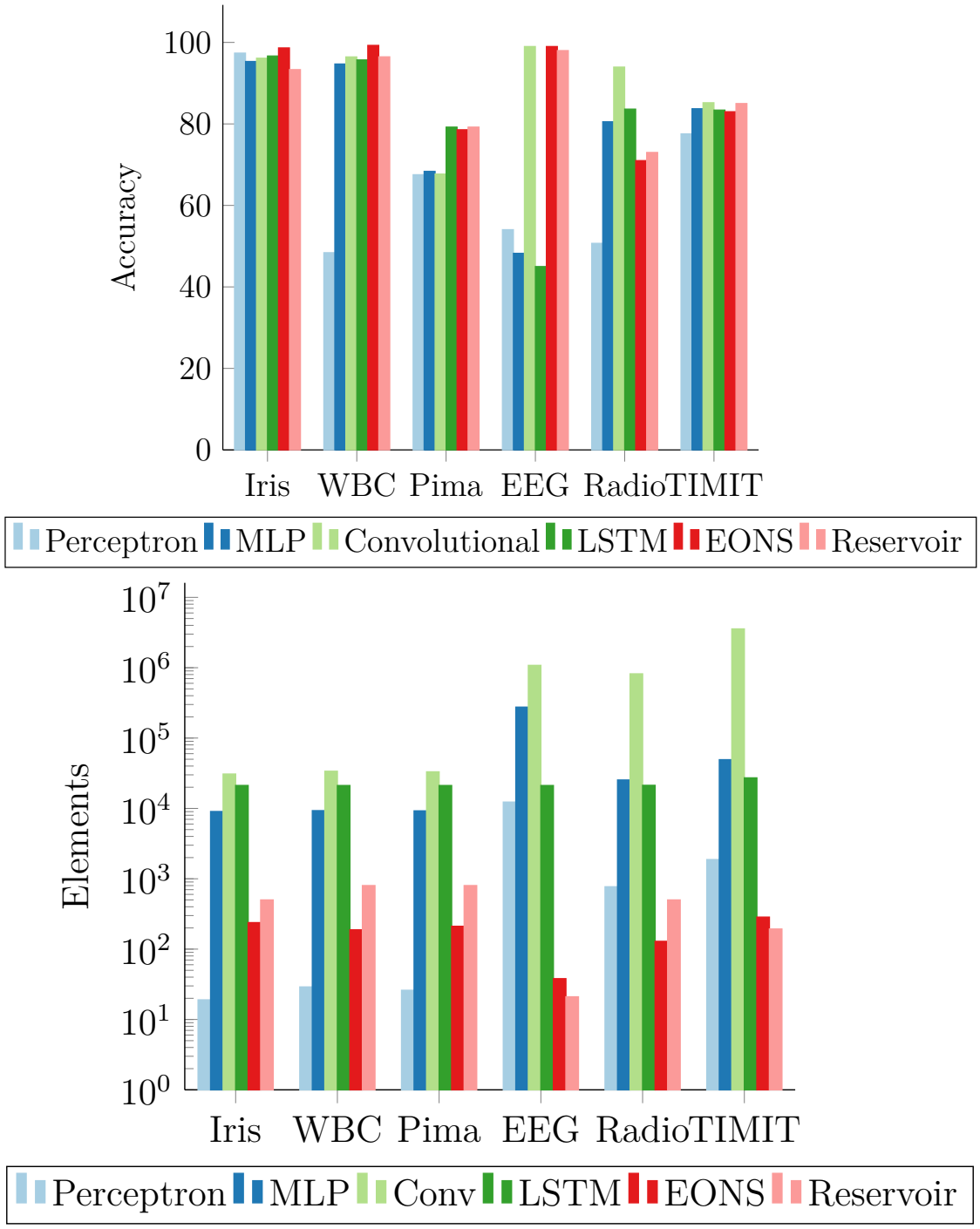


Figure 3.4: The top figure is a bar graph of model accuracy per data set. Accuracies are from table 3.3. The bottom figure is a bar graph of network sizes for comparison. Network sizes are in table 3.4

In Table 3.4, we display the neuron and synapse counts for the networks that produced the best classification accuracies. For LSTM, we define a "neuron" as an LSTM cell, which is more complex than a single neuron, but occupies the neuron's place in the neural network. For all of the Deep Learning models, we define a "synapse" as the number of trainable parameters. For all but the **Convolutional** model, this is in fact the number of synapses. For the **Convolutional** model, the actual synapse count is higher, because the convolutional and max-pooling layers train groups of synapses with a single parameter.

With the exception of the **Perceptron**, which does not classify as well as the others, the Deep Learning networks are orders of magnitude larger than the spiking, recurrent neural networks. Unsurprisingly, the **Convolutional** model has the most elements, peaking at roughly 20,000 neurons and 3,500,000 synapses for **TIMIT**. The spiking, recurrent networks are much smaller, highlighting their computational power. With **EEG** and **TIMIT**, we explored smaller reservoirs than the standard 200-neuron reservoir, and were able to generate reservoirs that were significantly smaller, yet retained the same classification accuracy. These are included in parentheses in Table 3.4 and represented visually as a bar graph in 3.4. It is worth noting that an individual synapse in the spiking recurrent networks is more complex than the individual synapses in the Deep Learning networks. However, since we are targeting building networks for spiking neuromorphic implementations, the synapses in the Deep Learning networks will still take up the same amount of "space" on the chip, even though they are not using the full functionality of the synapses that the spiking recurrent neural networks are.

Table 3.4: Neuron and synapse counts for the networks that produced the classification accuracies in Table 3.3.

Model / Data	**Iris**		**WBC**		**Pima**		**Radio**		**EEG**		**TIMIT**	
	N	S	N	S	N	S	N	S	N	S	N	S
Perceptron	7	12	11	18	10	16	258	512	4099	8194	626	1248
MLP	199	8835	203	9090	202	9026	450	24898	4291	270722	818	48450
Convolutional	775	30019	2678	31042	2314	30786	12278	805652	876547	202562	28006	3530652
LSTM	99	21091	98	21058	98	21058	98	21186	98	21058	98	27074
EONS	74	164	22	166	38	173	26	103	8	30	39	246
Reservoir	200	300	200	600	200	600	200	300	200 (7)	900 (14)	200 (53)	1000 (140)

3.6 Conclusion

We compare six approaches for training neuromorphic networks for six classification tasks. The six approaches comprised of four Deep Learning approaches, a spiking neural network training approach trained using evolutionary optimization (EONS), and a reservoir computing (RC) approach using spiking neural networks. We found that there was no one clear winner amongst the six approaches on all six tasks in terms of classification accuracy. However, for neuromorphic implementation, classification accuracy is often not the only metric considered; in particular, for certain applications, metrics such as size, weight, and power (SWaP) of the resulting neuromorphic implementation are also important to consider. As such, we also compared the sizes of the resulting networks for each of the six approaches. We found that, in general, the spiking approaches (EONS and RC) both produced networks that were orders of magnitude smaller than the networks produced by the successful Deep Learning approaches. Thus, networks trained using those approaches may be more suitable for certain SWaP constrained applications.

It is worth noting that in this work, we approached the comparison of the different methods naively in order to simulate how a non-machine learning expert would approach. There is much additional work that could be done to improve the performance of any given machine learning approach on any given application. Further, this work does not address significantly large datasets (hundreds of thousands or millions of examples) or datasets with many classes. The length of time training with the spiking neural network models (EONS, RC) needs to be compared to the amount of time to train non-spiking variants. We also suspect a comparison of training times between EONS and RC would yield interesting results. Other future work includes exploring extra models, such as Echo State Networks [45]. Because of the promising results for the spiking neural network approaches in terms of resulting network size, we intend to focus our research on those areas moving forward.

Chapter 4

Reservoir Computing and Genetic Algorithms in the TENNLab Framework

4.1 Overview

This chapter is reproduced work from a publication written by the thesis author [95]. Neuromorphic Computing is a burgeoning field of research. Many groups are exploring hardware architectures and theoretical ideas about spiking recurrent neural networks. The overarching goal is to exploit the low power promise of these neuromorphic systems. However, it is difficult to train spiking recurrent neural networks (SRNNs) to perform tasks and make efficient use of neuromorphic hardware. Reservoir Computing is an attractive methodology because it requires no tuning of weights for the reservoir itself. Yet, to find optimal reservoirs, manual tuning of hyperparameters such as hidden neurons, synaptic density, and natural structure is still required. Because of this, researchers often have to generate and evaluate many networks, which can result in non-trivial amounts of computation. This section employs the reservoir computing technique (specifically liquid state machines) and genetic algorithms in order to develop useful networks that can be deployed on neuromorphic hardware.

We build on past work in reservoir computing and genetic algorithms to demonstrate the power of combining these two techniques and the advantage it can provide over manually tuning reservoirs for use on classification tasks. We discuss the complexities of determining whether or not to use the genetic algorithms approach for liquid state machine generation.

4.1.1 Spiking Model

For this work, we use the DANNA2 neuromorphic model [78]. DANNA2 offers a software simulation and a digital hardware architecture for spiking recurrent neural networks (SRNN). Each network is a collection of neurons arranged within a two-dimensional coordinate system. Neurons follow a simple accumulate and fire model in which each neuron accumulates charge over time until a specified threshold is reached leading to an output spike. After a neuron fires, it enters a configurable refractory period in which it may not fire again until refractory is complete. Further, neurons have a configurable leak parameter. Spikes propagate from one neuron to another through synapses. Each synapse transfers a spike from one neuron to another with a configurable weight value and temporal delay. The weight parameter for a given synapse can be dynamically modified during operation using Spike Timing Dependent Plasticity (STDP). In this work, however, STDP is turned off.

4.1.2 Training: Evolutionary Optimization of Neuromorphic Systems

For our genetic algorithm, we use Evolutionary Optimization of Neuromorphic Systems (EONS), which has been employed previously to train SRNNs intended for use on neuromorphic systems [108, 90]. What makes EONS interesting is that it optimizes not only the numeric hyperparameters of the SRNN's, but also their structure. Networks in the EONS populations are represented as graphs in which the nodes and edges have optimizable parameters. Crossover and mutation operations leverage this graph-based representation. Each application must implement a fitness

function that takes a network as input and returns a single numerical score, in which higher fitness scores correspond to better performing networks. The EONS framework determines both the topology of the network (number of neurons and synapses and connectivity pattern) and the parameters of the network (weights, thresholds, etc.) Note that having to define a fitness function which determines what output comprises high success rates is different than that of a readout layer in reservoir computing.

4.1.3 EONS Applied to LSMs

We explore two methods for training liquid state machines: grid search and EONS. To use EONS, we must provide a value of fitness for each network in the population. The value of fitness is computed in multiple steps. First, the network is stimulated by applying spikes over time. This aids with static classification [4]. After the predetermined number of time-steps have passed, the state vectors are collected in the spike count format described in chapter 2. Then, the readout layer is trained. The fitness achieved by the readout layer is returned as the fitness value. Several other fitness values for genetic algorithms training liquid state machines have been explored, such as using metrics describing the network separability or spectral radius [44]. As we demonstrate in our results, relying on those properties may not be the best approach to finding successful reservoirs.

4.2 Experimental Results

The classification accuracies of reservoir computing on many well known datasets have been demonstrated in previous work by other researchers [4]. Here we focus on the Ionosphere dataset as a case study [114], as our goal is to demonstrate the usefulness of EONS when compared to grid search. First, we will describe the dataset. Then, we will delve into the experiment we performed. Finally, we will examine the results obtained with each of the individual methods.

4.2.1 Ionosphere

The Ionosphere dataset is a collection of information generated by a phased array of 16 high-frequency antennas. The antennas send a multipulse pattern targetting free electrons in the ionosphere. A receiver is activated in-between pulses. The result is 17 pulse numbers each composed of 2 complex attributes representing the electromagnetic signal, resulting in a total of 34 continuous valued features. The phase shift of returns is measured to determine the target velocity. In specific, an autocorrelation function is computed on the returned signal, which can be used to determine the Doppler velocity of the target. If a target is moving with constant velocity, the autocorrelation function will show a phase shift proportional to the pulse number. A sample is determined to be a “good" radar return if structure was detected (and the phase shift is proportional), and determined to be “bad" otherwise. Reservoir computing results from randomly generated reservoirs are previously reported as 92%, but an accuracy of 96% is typical of traditional machine learning algorithms, with a high of 98% [4].

4.2.2 Experiment Design

For our experiment we evaluated 60,000 networks using both grid search and EONS, which resulted in a total of 120,000 network evaluations. The number of tests was kept the same to demonstrate the results of each method with the exact same number of network simulations. There are 34 inputs to the reservoir, 20 outputs from the reservoir to the readout, and a softmax layer with two outputs for the readout. Each method was evaluated on the ionosphere dataset with a 50/50 train/test split. The readout was trained on each individual network for 1,000 epochs with a learning rate of 0.001 (determined experimentally).

We recorded the training and testing accuracy of each of the 120,000 evaluated networks as well as the metrics relevant to reservoir computing discussed in previous sections. Specifically, we look at the spectral radius of networks, which is related to the fading memory property. We also recorded the silhouette coefficient, which is a

commonly used metric in statistics ranging from -1.0 to 1.0. The silhouette coefficient represents the validity of clusters generated by clustering methods. A coefficient value of -1.0 means that all points are incorrectly clustered, and a value 1.0 represents fully valid clusters [99]. In the context of reservoir computing, it is a single value that can be viewed as a representation of the ability of a liquid to separate information, and it is computed from the state vectors generated by sampling a liquid after stimulation.

4.2.3 Grid Search Results

Knowing where to start with the hyperparameters for liquid state machine generation can be unclear. There are guidelines and recommendations for reservoir computing [65], but often researchers have to generate many random networks to find a satisfactory solution. To demonstrate this, we performed a grid search varying the number of hidden neurons and the synaptic density (relative to the number of hidden neurons). The number of hidden neurons was varied from 10 to 100 in increments of 10, and the synaptic density was varied from 5% to 50% in increments of 5%. For each of these configurations, 100 random networks were generated. After the coarse-grained grid search, the five best hyperparameter settings for number of hidden neurons and synaptic density were chosen, and a fine-grained grid search was performed in which we fine-tuned the search to vary across the best performing configurations in increments of 1 neuron and 1% synaptic density (also with 100 networks per configuration). In total, the coarse-grained grid search was 10,000 tests, and each fine-grained grid search was 10,000 tests, for a total of 60,000 tests.

The relevant results are shown in the heat maps in figures 4.1 and 4.2. The two heat maps in the first figure 4.1 are of the coarse-grained grid search training and testing accuracy. It is clear that certain configurations outperform others, with the highest synaptic densities resulting in far lower levels of accuracy. This is a result of the network activity becoming too chaotic to convey information. The success depends upon the combination of neurons & synaptic density rather than on one of the two parameters. In other words, this forms a joint probability distribution.

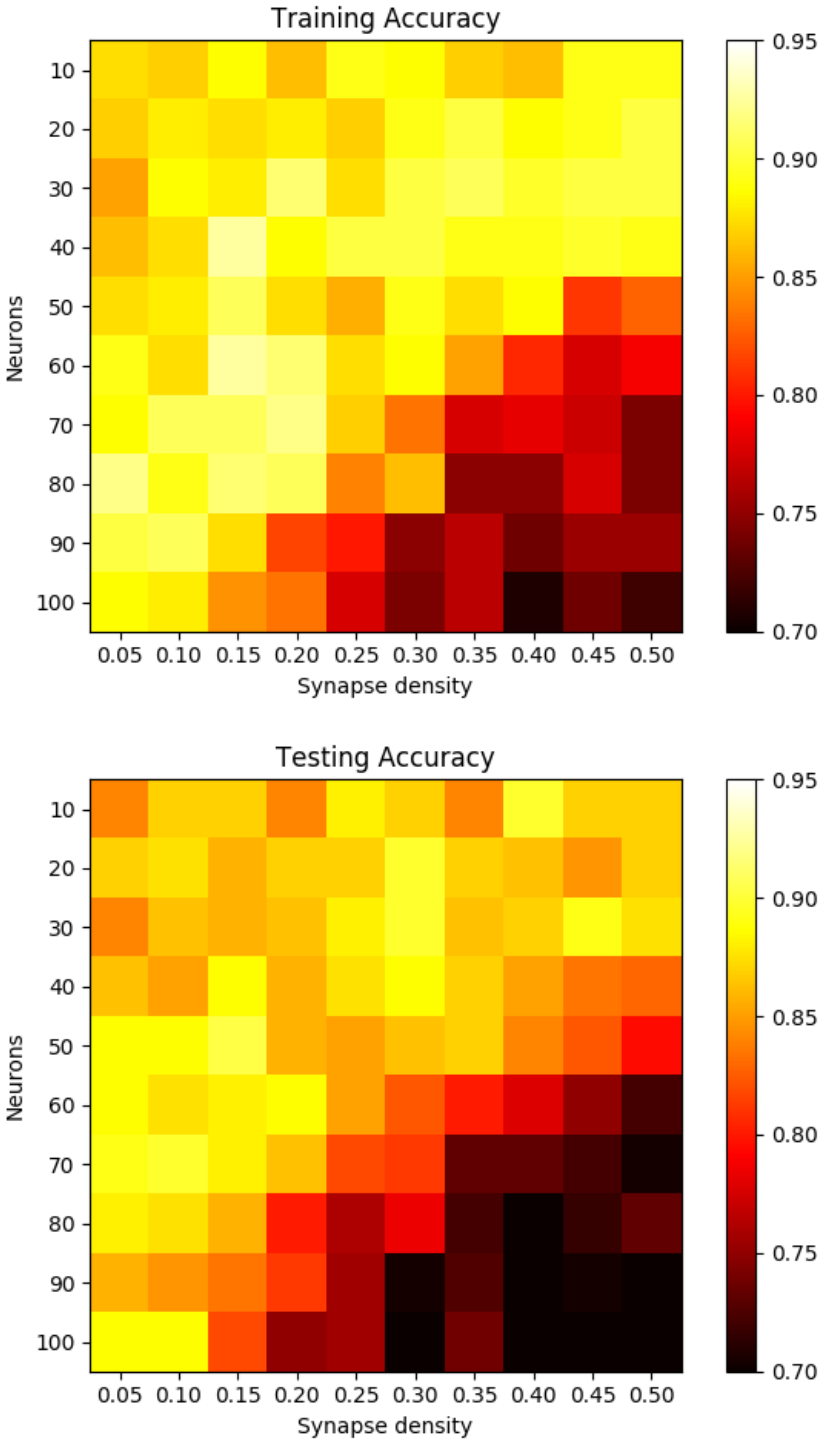


Figure 4.1: The heat maps above were generated from a coarse-grained grid search on randomly created reservoirs trained on the ionosphere dataset. Each block in the heat map represents the best accuracy from a population of 100. The number of hidden neurons and the synaptic density are varied on the y and x axes respectively.

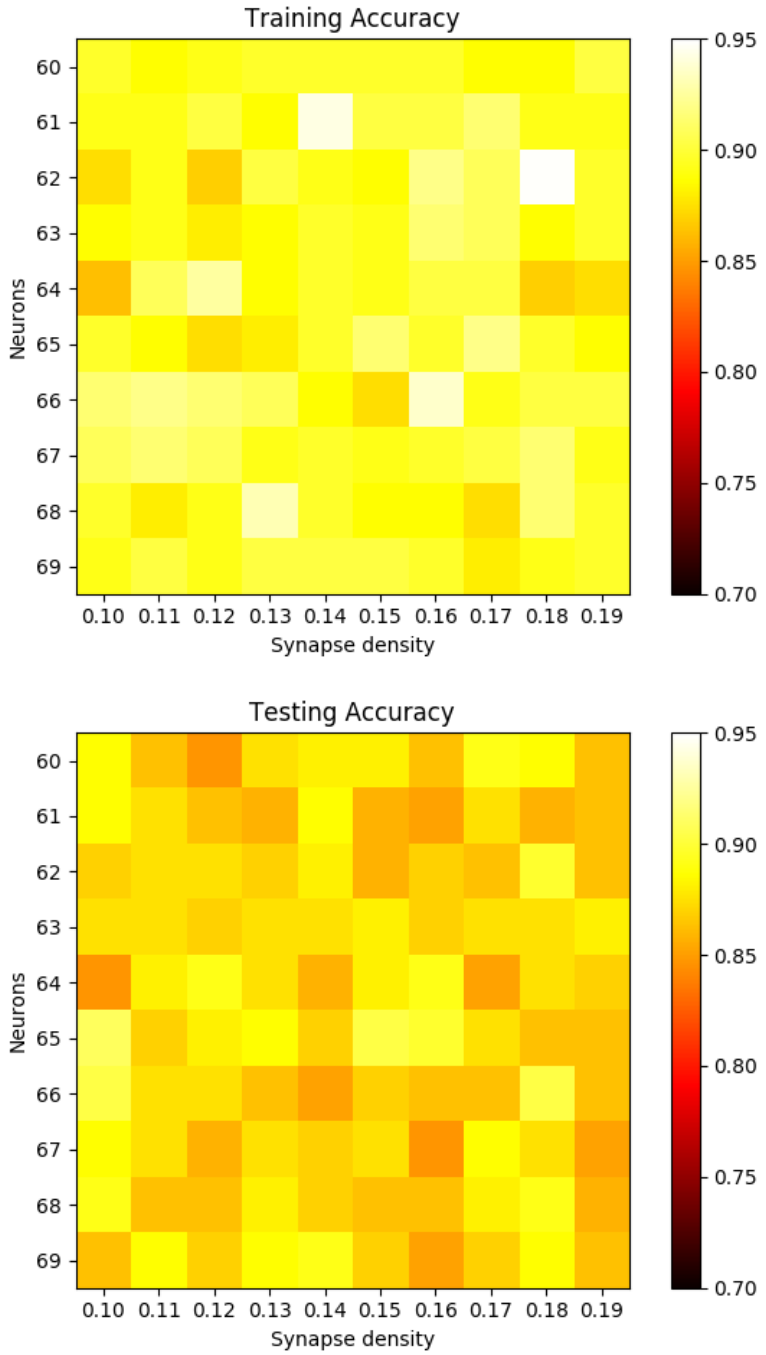


Figure 4.2: The heat maps above were generated from fine-grained grid search on randomly created reservoirs trained on the ionosphere dataset. Each block in the heat map represents the best accuracy from a population of 100. The number of hidden neurons and the synaptic density are varied on the y and x axes respectively. In particular, the fine-grained search is across 60 neurons centered on the best accuracy from the top level grid search: 0.15 synaptic density. This demonstrates that the hyperparameter configuration is an integral part of reservoir success.

To further demonstrate that hyperparameter configuration is important, we display results from the fine-grained search on the best training accuracy spot: 60 neurons centered on a synaptic density of 0.15. The two heat maps are of this fine-grained search 4.2, and show that high accuracy is attained across the spectrum of hyperparameters when an appropriate coarse-grained configuration is found. However, it is important to note that these heat maps are demonstrating the best accuracy achieved out of 100 networks for each configuration. Even after finding an appropriate selection of hyperparameters, the distribution of reservoir success can vary wildly. This is shown in figure 4.3, and serves as an example of the need to evaluate multiple reservoirs before finding a successful one. An exhaustive search is required in order to know the true distribution of successful reservoirs.

As mentioned earlier in this work, metrics associated with reservoirs are often evaluated in order to determine the validity and success of said reservoir. While there certainly can be a correlation between certain metrics and accuracy, having a “good" value for such a metric does not guarantee success. On the other hand, having high accuracy does not mean one will have good reservoir values. This is demonstrated in the box plots in figure 4.4. We can see that the average silhouette coefficient increases with accuracy, but still has a wide range of values. Further, the highest accuracy networks seemingly do not follow the trend, and thus appear to be outliers. Notice the lack of range on some of the x points, as they appear to be single instances of these high accuracy networks rather than repeatable scores. They are included because this data was all generated by the experiments performed as a part of this work, and the outliers demonstrate the necessity of hyperparameter tuning to achieve high levels of reservoir performance.

The spectral radius also narrows to a certain range with higher accuracy, but spectral radius values in that range exist across a majority of the spectrum of accuracy, so having a value in that range does not guarantee success [65]. Because the spectral radius alone is an inadequate measure of reservoir capability, and separability metrics all rely on gathering state vectors from reservoir simulation, it is essentially a requirement that the liquids be simulated for evaluating success accurately.

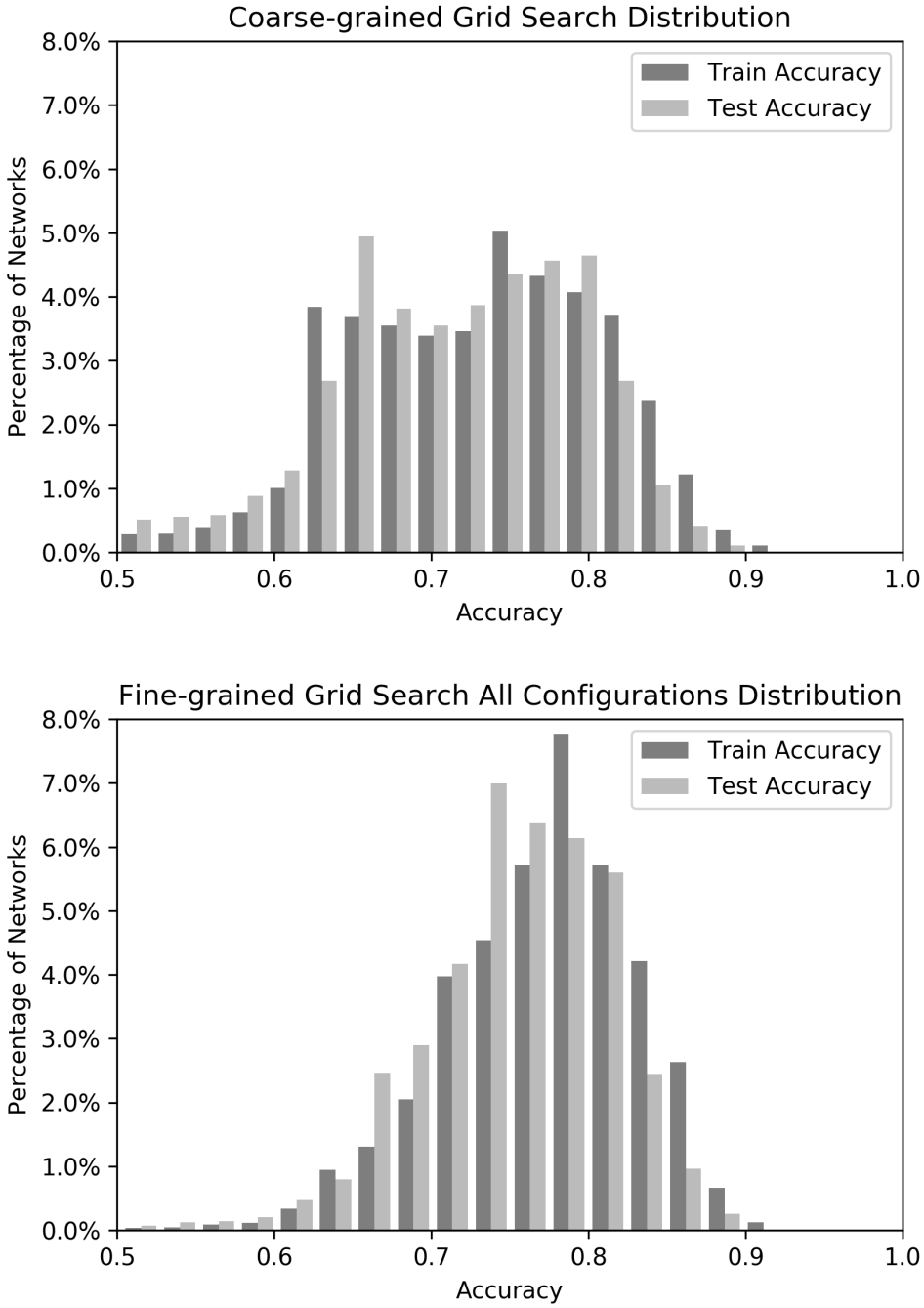


Figure 4.3: The histograms depicted above represent the accuracy distribution across configurations utilized in the grid search. It is clear that a valid choice of hyperparameters still does not ensure successful reservoirs. Further, even within a fine-grained search across a good choice of hyperparameters, many liquid evaluations may be required. The accuracy distribution is higher on average in the fine-grained approach, but the highest performing reservoirs are still a small portion of the overall distribution.

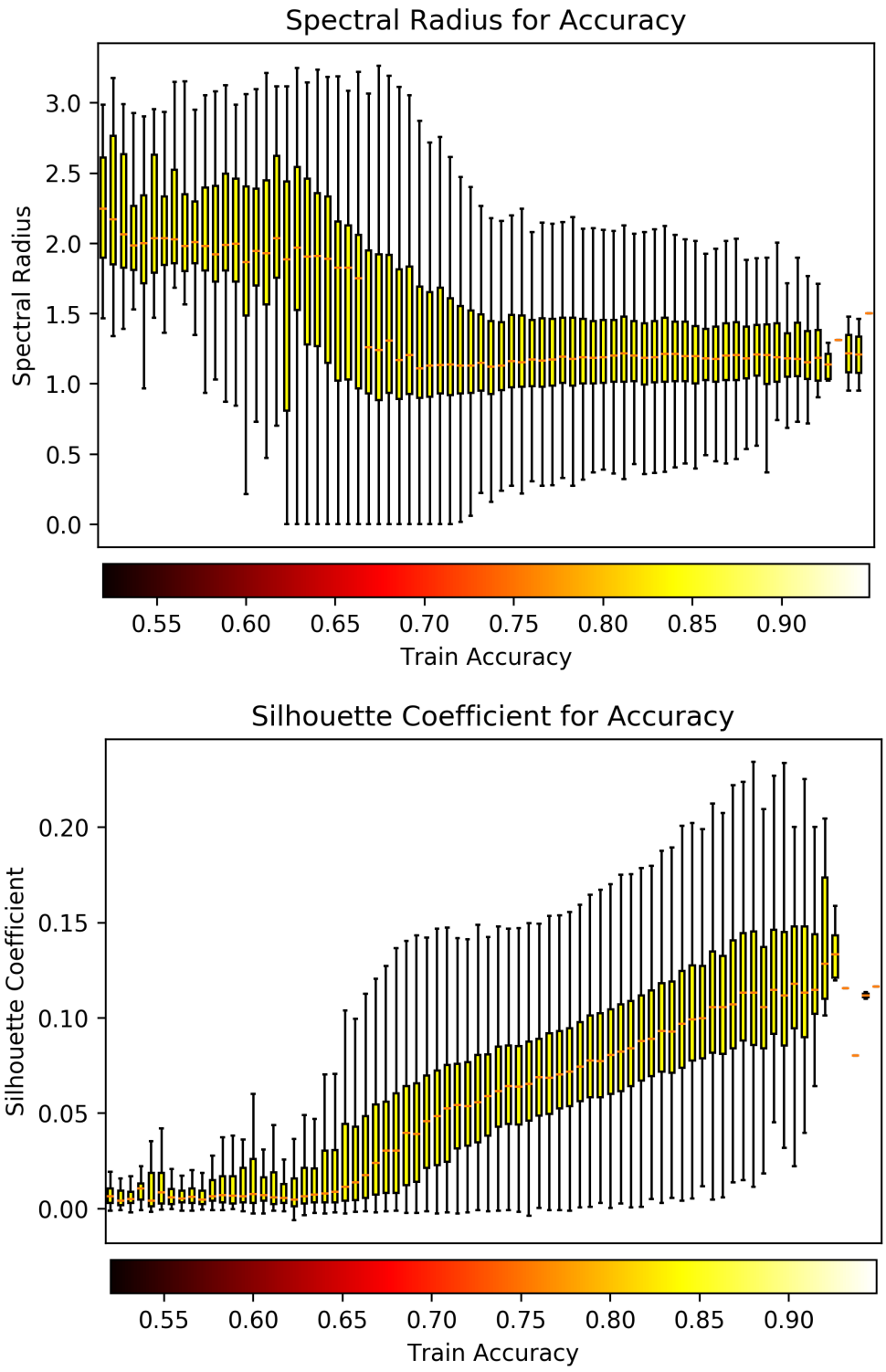


Figure 4.4: These graphs are generated from all 60k of the grid search tests. The average silhouette coefficient increases with accuracy, but still has a wide range of values. The spectral radius also narrows to a certain range with higher accuracy, but spectral radius values in that range exist across a majority of the spectrum of accuracy, so having a value in that range does not guarantee success, nor prevent it.

It is important to note that the rate of success from random generation can essentially go to zero if networks are not complex enough or too complex, even when the spectral radius falls into the recommended region from literature (which is less than 1.0, but not too low to not generate activity). The size of the network required depends entirely upon the problem attempting to be solved, and thus to truly map the distribution of rates of success from random generation for a particular problem, one would need to perform an exhaustive parameter sweep across the number of neurons, number of synapses, number of outputs being sampled, limiting delays on synapses, leak on/off, synaptic plasticity on/off, and any other adjustable network parameters. This is clearly comparable in complexity to utilizing a structured run of EONS.

4.3 Evolutionary Optimization of Reservoir Fitness

We set the population of each EONS run to be 50 networks. We performed 100 runs, each lasting 12 epochs, and each of which matched a different configuration from the coarse-grained grid search performed for random generation. In other words, one EONS run population was initialized randomly with 10 hidden neurons and 5% density, another was 10 hidden neurons and 10% density. That way, for each grid configuration, there was a matching configuration for a run of EONS. These parameters were selected to match the grid search, as 50x100x12 = 60,000, the same number of networks as the grid search. A correlation matrix of experiment variables from the EONS run is included in 4.5. The colors and values are based on the table 4.1. The spectral radius relationship with synapses is demonstrable with the correlation matrix. Neurons are also relevant, but less so. The silhouette coefficient is also clearly correlated with the training and testing accuracies.

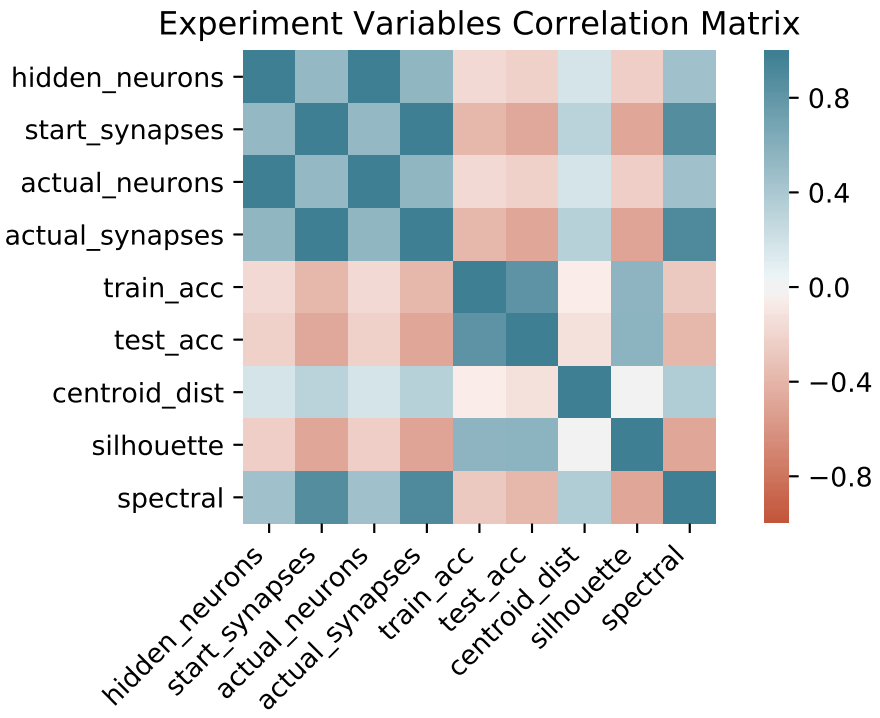


Figure 4.5: Correlation matrix of variables recorded throughout the reservoir generation process.

Table 4.1: Correlation matrix of associated recorded reservoir variables.

	h_n	s_s	a_n	a_s	tr_a	te_a	c_d	sil	spec
h_n	1	0.529	1	0.542	-0.172	-0.224	0.182	-0.242	0.461
s_s	0.529	1	0.529	0.993	-0.384	-0.474	0.320	-0.493	0.880
a_n	1	0.529	1	0.542	-0.172	-0.224	0.182	-0.242	0.461
a_s	0.542	0.993	0.542	1	-0.387	-0.482	0.332	-0.507	0.900
tr_a	-0.172	-0.384	-0.172	-0.387	1	0.827	-0.047	0.556	-0.266
te_a	-0.224	-0.474	-0.224	-0.482	0.827	1	-0.129	0.572	-0.389
c_d	0.182	0.320	0.182	0.332	-0.047	-0.129	1	-0.012	0.367
sil	-0.242	-0.493	-0.242	-0.507	0.556	0.572	-0.012	1	-0.489
spec	0.461	0.880	0.461	0.900	-0.266	-0.389	0.367	-0.489	1

In order to perform a direct comparison of the success between the grid search and the EONS runs, we sorted all 60,000 tests for each method by training accuracy. Then, the top 500 liquids were taken for each method (based on training accuracy), and the distribution of accuracy was plotted. Figure 4.6 demonstrates that although both methods are capable of reaching expected levels of accuracy, the structured search performed by EONS skews the distribution towards higher performing networks. Further, looking at figure 4.7, we can see that EONS converges to the expected accuracy regardless of the beginning population initialization configuration, which is in contrast to figure 4.8. In other words, even though each EONS run began with a different number of hidden neurons and synaptic density combination, they all converge. This fits our thesis, which is that simply allowing an intelligently structured search from EONS to generate reservoirs arrives at successful hyperparameters regardless of the starting condition, which, as demonstrated, does not hold for random generation.

4.4 Discussion

The number of neurons in a system has an effect on the complexity that a neural network can represent. For reservoir computing, knowing exactly how many neurons are necessary to adequately separate one's problem set does not have an answer. Choosing to use too many can result in needlessly large networks, while choosing too few can easily result in networks that are incapable of separating the information. The properties of simulation and evaluation of SRNNs are dependent on many different factors. For instance, higher levels of connectivity typically generate more events, and are thus slower to simulate. Further, the number of neurons being sampled for output can contribute to the overall success of reservoir systems. In order for random generation of LSMs to succeed, one must be randomly generating networks with sufficient complexity. Certainly, one may perform a search over the many different parameters, but this can end up taking just as much time as EONS and perform comparably in terms of accuracy.

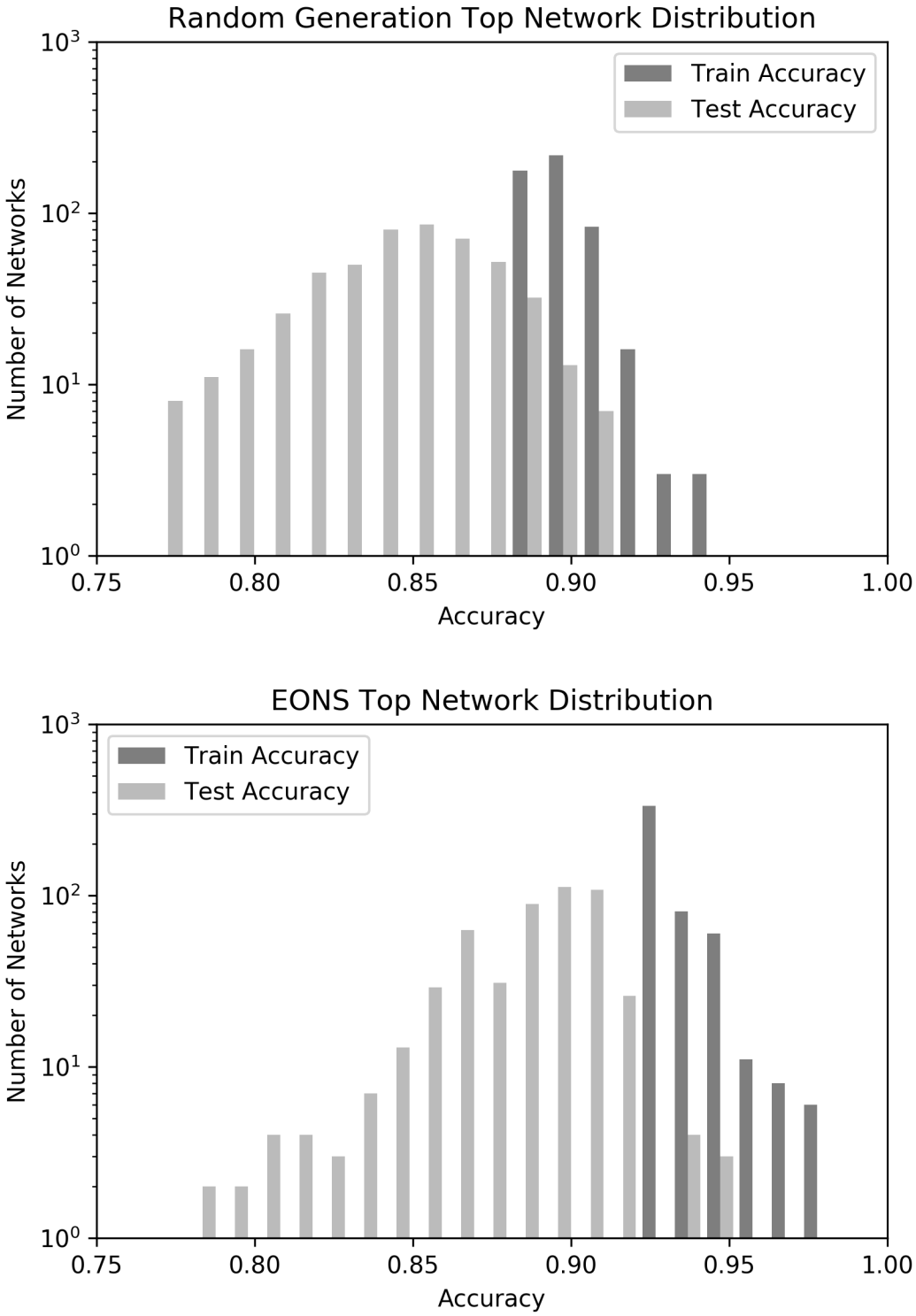


Figure 4.6: Top 500 performing networks from each of the methods. The y-axis is presented on a log scale. The plots demonstrate that the EONS results are skewed towards higher performance. Overfitting can be remedied through manual tweaking of parameters, and is less apparent when reviewing more than the top 500 performers.

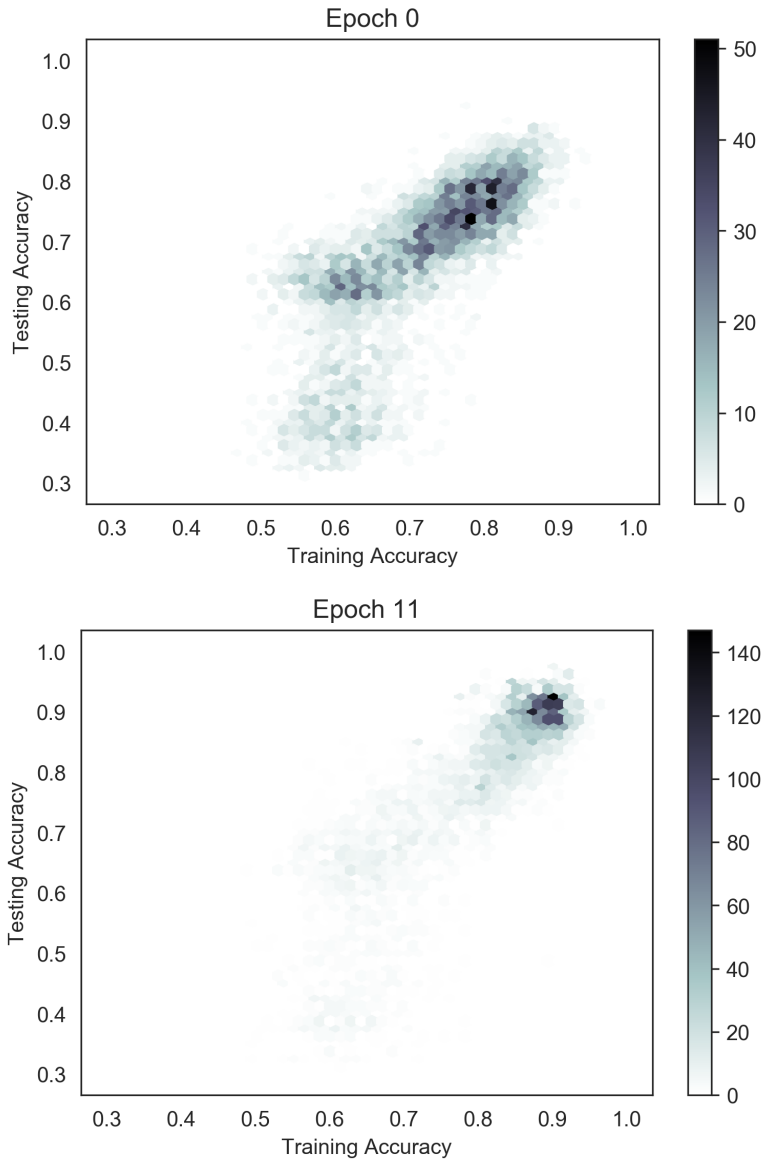


Figure 4.7: The top two figures are hex-bin plots of the EONS tests performed. The top-left plot is epoch 0, and the top-right plot is epoch 11. Each plot is of all 50 networks from all 100 runs for a total of 5k. Each hexagon represents a bin, and all of the samples falling into that region are integrated into that bin. The color gradient represents the number of samples in a bin. EONS converges to the expected accuracy (approx. 90%) regardless of the starting configuration, which is an inherently different result than the two plots at the bottom; the left being the coarse-grained grid search, and the right being of all 5 fine-grained grid searches.

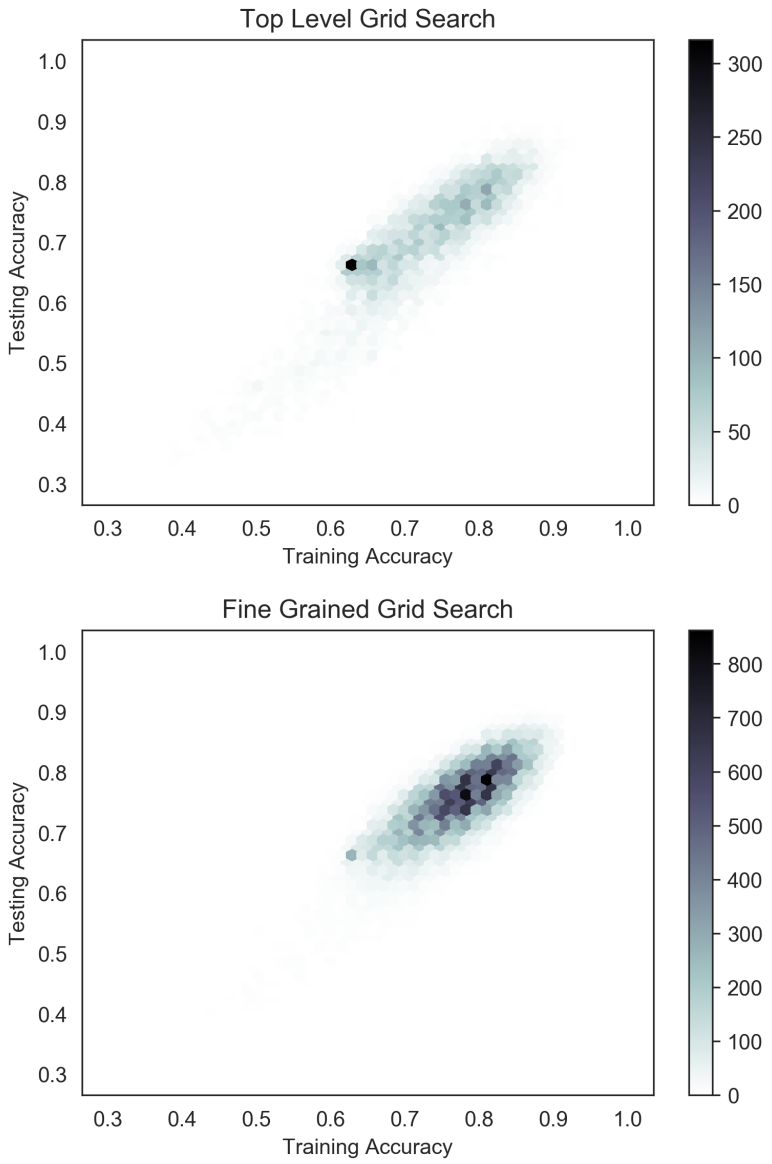


Figure 4.8: The top two figures are hex-bin plots of the EONS tests performed. The top-left plot is epoch 0, and the top-right plot is epoch 11. Each plot is of all 50 networks from all 100 runs for a total of 5k. Each hexagon represents a bin, and all of the samples falling into that region are integrated into that bin. The color gradient represents the number of samples in a bin. EONS converges to the expected accuracy (approx. 90%) regardless of the starting configuration, which is an inherently different result than the two plots at the bottom; the left being the coarse-grained grid search, and the right being of all 5 fine-grained grid searches.

As demonstrated in this work, generating satisfactory reservoirs for a given problem can depend on prior knowledge, which often leads to researchers generating many random networks before arriving at one that works. In many scenarios, we would recommend that systems like EONS be put into place, as they can more efficiently search through that same problem space.

4.5 Conclusion

In this work, we have explored reservoir generation for a spiking recurrent neural network implementation of liquid state machines on a classification problem. We explored two well-known hyperparameter optimization techniques – a two-level grid search, and a genetic algorithm called EONS, which includes network structure of the reservoir in its optimization. In our tests, the EONS optimization discovered better networks on the whole, and also converged more reliably. We used our experiment to evaluate the metrics of spectral radius and silhoutte coeficient on reservoir effectiveness. While the two metrics showed converging trends for networks that train better, when used in isolation, they are not sufficient to be used to construct good reservoirs.

Chapter 5

Reservoir Computing for LIDAR Control Applications

5.1 Overview

Control systems modeling and optimization is a topic of interest across many different fields. There are a variety of approaches useful for developing strategies and approximating optimal policies. The TENNLab research group has primarily focused upon the use of genetic algorithms and spiking neural networks for control applications. Another valid and widespread approach is the practice of Reinforcement Learning [123, 51]. The study of reinforcement learning has seen much success in recent years, particularly in the context of Deep Reinforcement Learning [80, 81]. One of the key insights behind modern Reinforcement Learning is the ability to train a control application with gradient descent. Further, some works have performed brief explorations of reinforcement learning in conjunction with different feature transformations, namely, reservoir computing [7, 92]. In this chapter, we will discuss the combination of reinforcement learning and reservoir computing when applied to the TENNLab group's suite of LIDAR control applications.

5.2 Reinforcement Learning

Reinforcement learning is based on a simple premise: learn based on experiences and the positive or negative feedback obtained within that system. In other words, it is similar to how humans learn behaviors through experiences in real life. The specific terminology describes an agent (your actor) in an environment (game/world) that takes actions (makes decisions) and receives feedback (a reward). The goal of reinforcement learning algorithms is to maximize the cumulative reward based on actions the agent learns to take within the given system.

There are several different variants of reinforcement learning. Q-Learning and Policy Gradients are two of the most prominent versions [123]. The reinforcement learning process is generally modeled as a Markov Decision Process with several different parameters: environment, agent, states for those respective objects, transition probabilities, and a reward provided to the agent after a transition from one state to another is made. Generally, the idea is to learn a policy for decision making: given that the agent currently finds itself in state s, what action does the learned policy dictate the agent should make? Other parameters can be included within this formulation, but some variant of the aforementioned parameters always exist. These ideas are usually denoted as the following:

- **Actions:**

  $\boldsymbol{A}$: A set of "actions" the agent can take, E.g. left, right, up, down.

- **States:**

  $\boldsymbol{S}$: A set of states the environment and agent can exist within.

- **State Transition Probabilities:**

  $\boldsymbol{P}_a(s, s')$: Probability of action a causing a state change from state s to s'.
  $\boldsymbol{P}_a(s, s') = Pr(s_{t+1} = s' \mid s_t = s, a_t = a)$

- **Reward:**

  $\boldsymbol{R}_a(s, s')$: A value associated with the state transition given a selected action, a. E.g. points gained after choosing to jump in a game.

- **Policy:**

  $\pi(a, s)$: A function mapping states to actions.

  $\boldsymbol{\pi} : A \times S \rightarrow [0, 1]$

  $\pi(a, s) = Pr(a_t = a \mid s_t = s)$

In order to learn a policy, algorithms traverse the Markov Chain while recording rewards associated with each transition. They then update their policy generation depending on the rewards achieved. This "policy update" depends on the type of algorithm. For instance, Q-Learning involves learning the state transition probabilities empirically, and the policy uses these probabilities directly. Policy gradients, on the other hand, attempt to optimize the policy directly without considering the expected reward given a state. The full details behind the mathematics of policy gradients may be found in the famous Richard S. Sutton and Andrew G. Barto Reinorcement Learning textbook [123].

In order to explore the combination of reservoirs and reinforcement learning, the readout layer described in previous chapters was modified to allow for usage as a stochastic policy network. A stochastic policy network is one in which the actions to be performed in the control loop are sampled from a probability distribution generated by the neural network. The distribution utilized depends on several factors, perhaps the most important of which being whether the action space is continuous or discrete. If the action space is discrete, the distribution is usually a Boltzmann distribution generated by a softmax function. On the other hand, when dealing with a continuous action space, the network can be designed to describe a normal distribution through mean and standard deviation values output by the network. The continuous value can then be sampled from this distribution. In the case of this work, the neural network is the readout layer described throughout earlier chapters, a multinomial logistic regression (linear combination of features with a softmax transfer function). This

works well for our case because the TENNLab LIDAR applications can be described within a discrete action space.

5.3 Policy Gradients

The policy gradient theorem allows for direct optimization of a given policy function. The mathematics underlying policy gradients lead directly to the implementation used in this work, so it will be useful to include a brief review and discussion of the technique.

The policy, as introduced in the previous section, is defined by the following equation:

$$\pi(a, s) = Pr(a_t = a \mid s_t = s) \tag{5.1}$$

Policy gradient methods are concerned with directly optimizing the policy. Therefore, an objective function must be defined such that the success of a given policy is adequately gauged. An example of such a score function can be seen in the following equation:

$$J(\theta) = E_{\pi\theta}\left[\sum \gamma r\right] \tag{5.2}$$

It is essential to note the reward value contained within this definition of the policy score function. The expected value of a given policy depends directly upon the achieved rewards. In fact, one can see the necessity of good rewards when the summation is unrolled into a full trajectory.

$$J_1(\theta) = E_\pi\left[R_1 + \gamma R_2 + \gamma^2 R_3 + \cdots\right] = E_\pi(V(s_1)) \tag{5.3}$$

The above equation represents the policy score for trajectory one. In other words, the policy score function is defined by the expected value of utilizing a policy parameterized by θ starting in state s_1. The θ parameter, in the case of our readout layer, is defined by the multilayer perceptron weights. The policy score function

gives us a differentiable function that we can attempt to optimize. Therefore, the parameter updates can be defined by following the gradient of the score function.

$$\theta \leftarrow \theta + \alpha \nabla_\theta J(\theta) \tag{5.4}$$

Taking the gradient of the score function entails a bit of mathematical trickery, and thus it is not necessary to cover the full derivation. However, the ultimate result is vital for understanding how the readout layer reinforcement learning implementation works. Ultimately, after taking the derivative of the score function, one ends up with an equation for updating the theta parameter by directly taking the partial derivative of the log of the policy with respect to the network weights and multiplying it by the reward trajectory. In this case, the policy is the softmax function.

$$\Delta\theta = \alpha * \nabla_\theta log\pi(s,a)R(\tau) \tag{5.5}$$

The softmax function provides a boltzmann distribution, which is excellent for probabilistic interpretations of values. However, reinforcement learning can suffer from the exploration/exploitation trade-off if sufficient work is not done to negate this effect. A typical solution for exploration/exploitation trade-off with softmax involves adding a temperature parameter for smoothing the distribution [123, 94, 40, 32]. As the temperature approaches infinity, all outcomes become equally likely. As the distribution approaches zero, the policy instead becomes greedy. The softmax function with temperature is in the following equation.

$$\sigma z_i = \frac{e^{\frac{z_i}{T}}}{\sum_{j=1}^{K} e^{\frac{z_j}{T}}} \tag{5.6}$$

The implementation of policy gradient reinforcement learning can be seen in appendix A. The function in question is UpdateWeightsREINFORCE, which provides an implementation of the simple and well known REINFORCE [139] algorithm making use of the above parameter update equation. Given this understanding of policy gradients, let us now delve into the control applications upon which it is applied.

5.4 TENNLab LIDAR Control Applications

The TENNLab software framework contains implementations of several different control applications [88]. Many of these applications use LIDAR sensors for remote sensing with pulsing lasers. TENNLab has applied EONS directly to the LIDAR applications in the past with varying success. A number of the applications are implementations of Atari games, but with LIDAR sensors used for information retrieval. This allows for indirect comparison with the OpenAI gym's suite of Atari environments [80]. The work in this dissertation focuses on three separate control problems: Asteroids 5.1, Spaceinvaders 5.2, and Bowman 5.3. These three applications differ in an important manner: movement. Asteroids allows the player to move freely throughout the game environment. Spaceinvaders restricts the player to the x-axis along the bottom of the screen, which requires strafing back and forth as the only option for movement. Finally, the player in Bowman is stuck in place. The player aims the angle of the bow in order to shoot birds with arrows. Thus, the Asteroids player moves freely, the Spaceinvaders player strafes, and the Bowman player attempts to angle the bow for points. All three are discussed at length in a previous publication [88]. A brief description of each application is included in this section.

5.4.1 Asteroids

The Asteroids Atari game is iconic and well known. A ship is placed in space and has to avoid/destroy Asteroids. The Asteroids are floating around in various directions. When an asteroid is shot, it breaks apart into multiple smaller Asteroids unless it is already the smallest version. The TENNLab framework includes an implementation of the Asteroids game but equips the player's ship with LIDAR sensors that detect Asteroids in the field. The number of LIDARs on the ship is configurable. For this work, the number is configured to thirty. An example screenshot of the simulation is included in figure 5.1. Players are able to rotate left, rotate right, thrust, and fire,

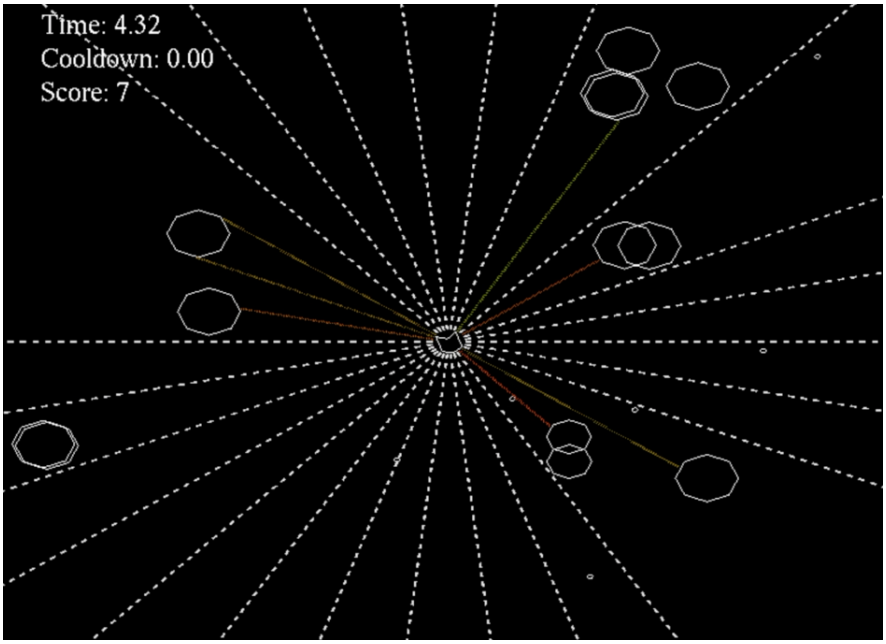


Figure 5.1: An example screenshot of the TENNLab Asteroids simulation. The LIDAR sensors are represented by dotted lines emanating from the player's ship. The color of the line represents the distance of a detected object.

as well as any combination of those three choices or none of them at all. The game length is limited to three hundred time steps.

5.4.2 Spaceinvaders

The Spaceinvaders Atari game is somewhat similar to Asteroids save a few distinctions. Significant distinctions include the limited movement of the player ship, which is limited to the x-axis at the bottom of the screen. The enemies, named invaders, spawn at the top of the screen and move towards the bottom of the screen. The game is over when an invader reaches the bottom of the screen. The length of these games is not limited. An example screenshot of the Spaceinvaders simulation is included in figure 5.2. The TENNLab implementation of Spaceinvaders includes LIDARs attached to the player. The colors represent the distance of a detected invader. These distances are then fed into spiking neural networks as the game state.

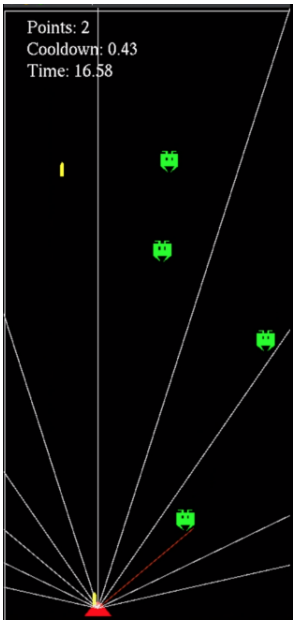


Figure 5.2: An example screenshot of the TENNLab Spaceinvaders simulation. The LIDAR sensors are represented by lines extruding from the player's ship. The color of the line represents the distance of a detected object.

5.4.3 Bowman

Bowman is the only application of the three that is not an original Atari game. Instead, it is derived from a popular internet flashgame by the same name. The goal of this application is to angle a bow to shoot as many birds as possible in a limited amount of time. The time limit, in this case, is 60 game time seconds. Each time step in-game lasts 0.015 seconds. An example screenshot from the game environment is provided in figure 5.3. The bow is equipped with LIDAR sensors that change color based on the distance of the detected object. The distance of detected objects is given to the spiking network as the current game state.

5.5 Experiment Design

The goal of the experiment is to design liquid state machines that provide significant feature transformations, so that they may successfully guide the control applications using reinforcement learning. A major question to address is the importance of the

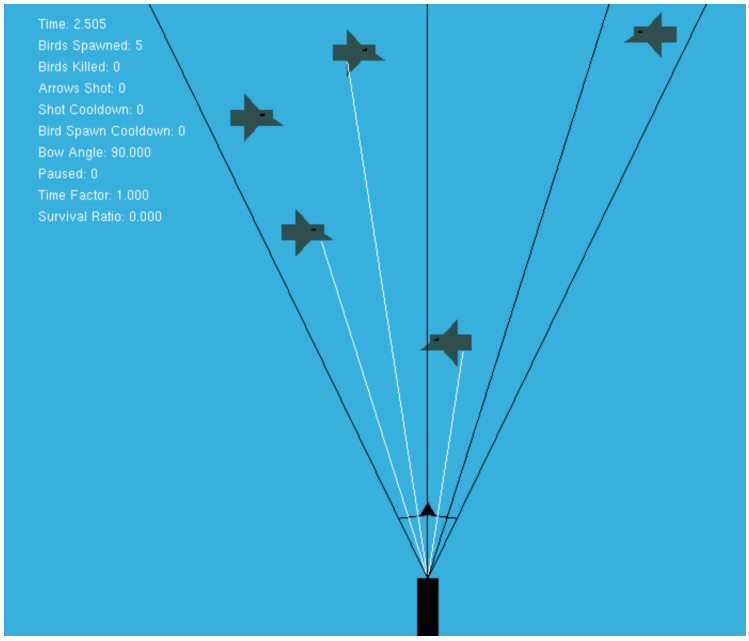


Figure 5.3: An example screenshot of the TENNLab Bowman simulation. The LIDAR sensors are represented by lines emanating from the player's ship. The line turns white when and object is detected.

liquid for feature transformation. We address this question by implementing policy gradient-based reinforcement of a reservoir computing system with various EONS-generated reservoirs, and of a system without a reservoir – where there is only a readout layer to be trained.

It should be noted that there are some control applications that are simple enough that a trained readout can solve them. One example is the "cartpole" problem, where a pole must be kept balanced on a cart that may be pulsed left or right. There is an implementation of this simulation within TENNLab [102] and within OpenAI Gym [80]. The single layer policy gradient was implemented in both TENNLab (using c++) and OpenAI Gym (using Pytorch). In Figure 5.4, we show results of training a readout layer using policy gradients on both simulation implementations. In both cases, the problem is simple enough that the readout is sufficient. In our work below, we include the readout-only baselines as a way to evaluate whether the reservoir is adding significant feature transformations to the system.

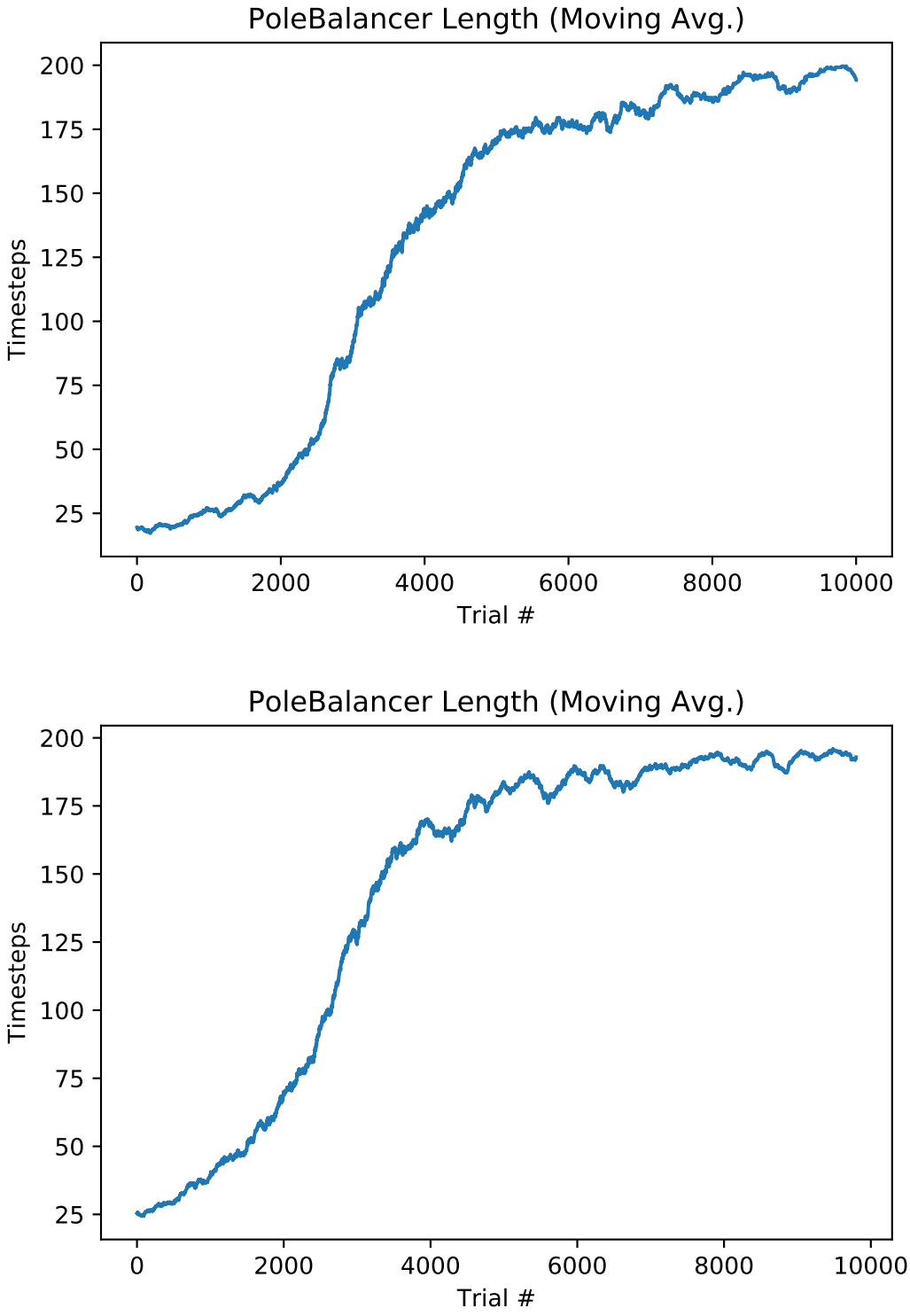


Figure 5.4: Polebalance results readout only. The top is of the readout trained on the TENNLab application, and the bottom is trained on OpenAI gym's implementation. The success serves as a baseline for policy gradient success without a reservoir.

When applying reservoir computing, we generated reservoirs with EONS, as described in the previous chapter. The fitness functions are more complex, however. The methodology for performing EONS was as follows. For each reservoir, we trained the readout for 1000 epochs, and used the final readout's success on the game as the fitness function. We performed 30 independent EONS runs with populations of ten networks, for 12 hours each. Then, for the best network in each run, we trained the readout for 10,000 epochs (9,100 for spaceinvaders, truncating it because of the long simulation times). We applied this methodology to make the best use of our CPU resources – as a readout layer becomes more successful in solving a control application, the simulation time of the application becomes the dominant time overhead. This is why we limit the readout training during EONS, but expand upon it once we select our final network.

In order to enable valid exploration of the environment, we had to experiment with the temperature variable in both Asteroids and Spaceinvaders. Asteroids experimentally worked best with a temperature value linearly decaying from 5,000 to 1 at a rate of 1 per episode, whereas Spaceinvaders used a temperature value decaying from 100 to 1 at a rate of 1/90 at each episode. No temperature value was required in the Bowman application.

5.6 Results

The results of the experiment are varied. We explore the results of each application, as they all differ.

5.6.1 Asteroids

For each section, we present trajectory graphs for EONS, and then training graphs for the readout layers. The score function for Asteroids is defined as $\frac{Asteroids_destroyed^2}{projectiles_fired}$, which attempts to reward accuracy as well as a pure score based on the number of asteroids destroyed. The fitness trajectories for the 30 EONS runs are displayed in Figure 5.5. As shown on the graph, the trajectories vary widely, with some never

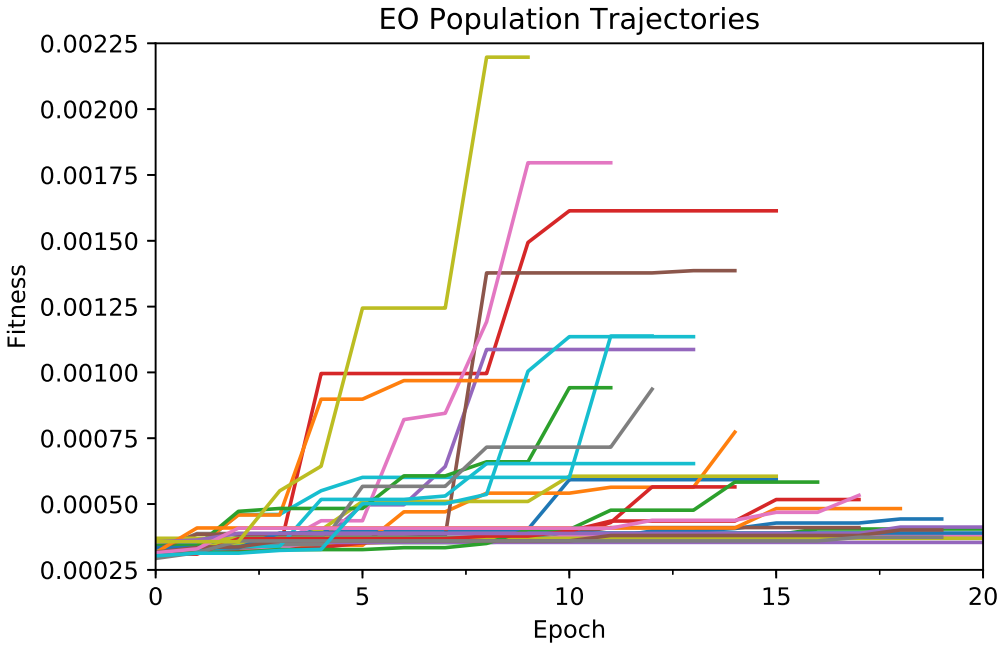


Figure 5.5: Trajectories of individual EONS runs for Asteroids

achieving good fitness. One run clearly had the best network, and it is this network that we use for the longer readout training. When a run starts to optimize higher fitness values, the application simulation starts to dominate the running time. This is because the application is typically running for longer periods of time (the player is not dying quickly). Since we cap each EONS run at 12 hours, this means that the more successful runs go through fewer epochs. Clearly, were we to run EONS for longer, and with more independent populations, we would expect to find better networks.

In Figure 5.6 we plot two metrics of training the readout, both in the non-reservoir case (labeled "Readout") and with the best reservoir from the EONS run. There are several interesting features from these graphs. First is the impact of the temperature setting on the reservoir. The temperature has been set so that the training explores more randomly in the first half of the runs, but then it hones in on readout weights in the second half. This results in better scores in the game, as displayed in the top graph.

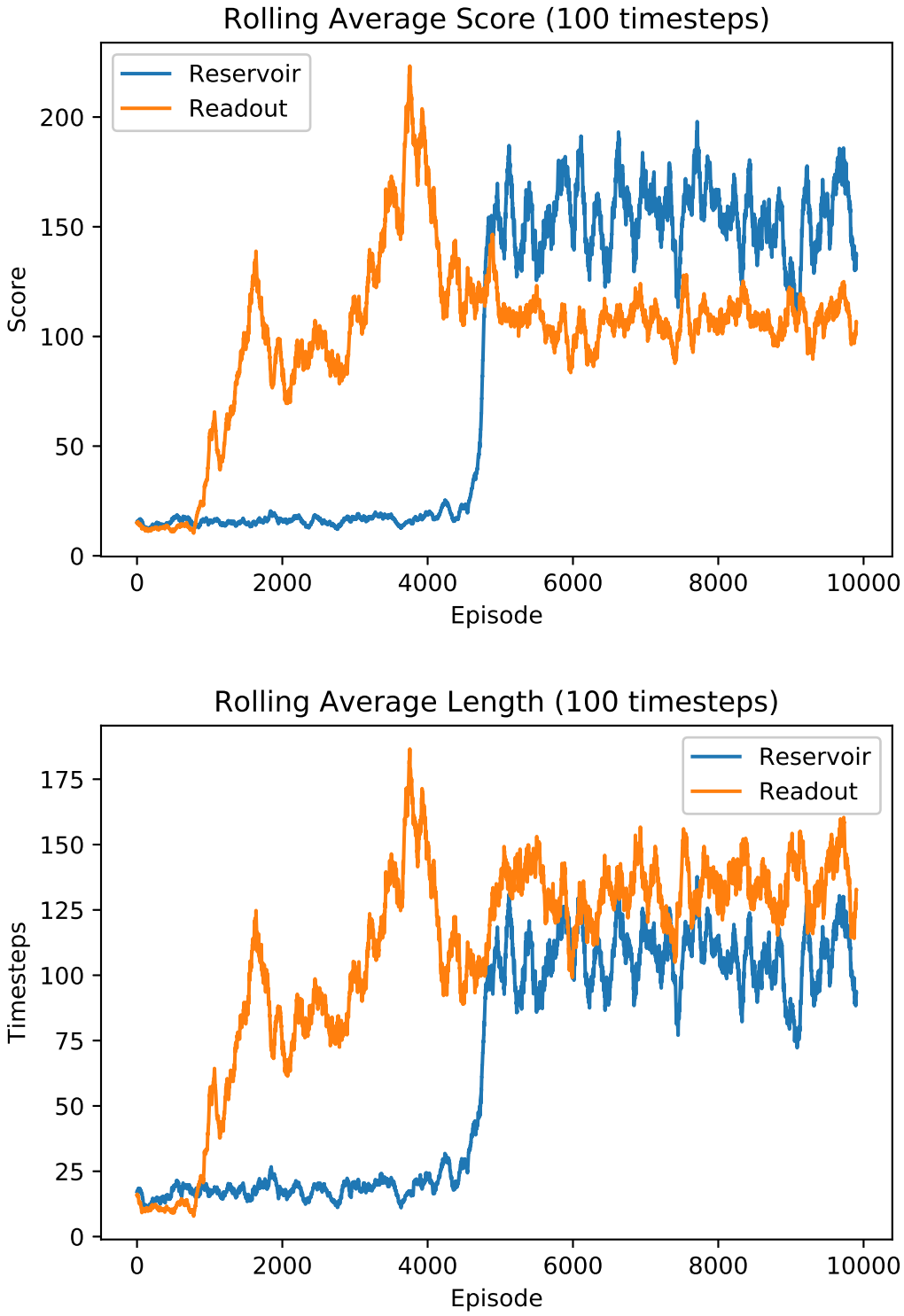


Figure 5.6: The best network taken from the EONS run was trained for an extended period of time. The graphs above show how behavior changed throughout they training epochs.

The readout-only training does not converge on higher scores, and in the second half of the training, it converges to weights which perform more poorly than with the reservoir. We conclude that the reservoir has a positive effect on the training. The readout-only training may possibly be improved with regularization [56], but the reservoir obviates this need.

The bottom graph shows the average duration that the player is alive when training the readout. Although this duration is not a fitness metric, it is an interesting metric to explore. Although the readout-only system scored lower than the reservoir system, its player lived decidely longer. The reason is that it trained its player to both move and shoot, and was able to evade asteroids. The reservoir system, on the other hand, did not have the player move, but instead trained it to shoot more accurately. As a result, it achieved higher scores, but its player did not live as long. Figure 5.7 shows the fitness scores of the two training runs. This shows the effect of temperature in the reservoir very clearly, following the same pattern as in Figure 5.6.

Finally, Figure 5.8 shows the probability distribution of the output events for Asteroids, with and without the reservoir, while training the readout. There are three separate output events: Rotate left/right, thrust and fire. Each of these has a "no action" option, so there are 12 combinations of actions that may be taken at any time. The figure confirms that the readout alone trains to be more active than it does with the reservoir. A possible avenue of future reseach is to modify the fitness function to encourage certain actions, or certain combinations of actions, for example to encourage the player to move and track asteroids. Of course, that adds yet another hyperparameter to the exploration (see the discussion in section 5.7 below).

5.6.2 Spaceinvaders

With Spaceinvaders, the length of if the game is tied directly to the score, as failure to kill an invader before it reaches the bottom of the screen results in immediate game termination. The fitness function is defined by $\frac{invaders_killed}{number_of_shots}$ to reward both longevity and efficiency.

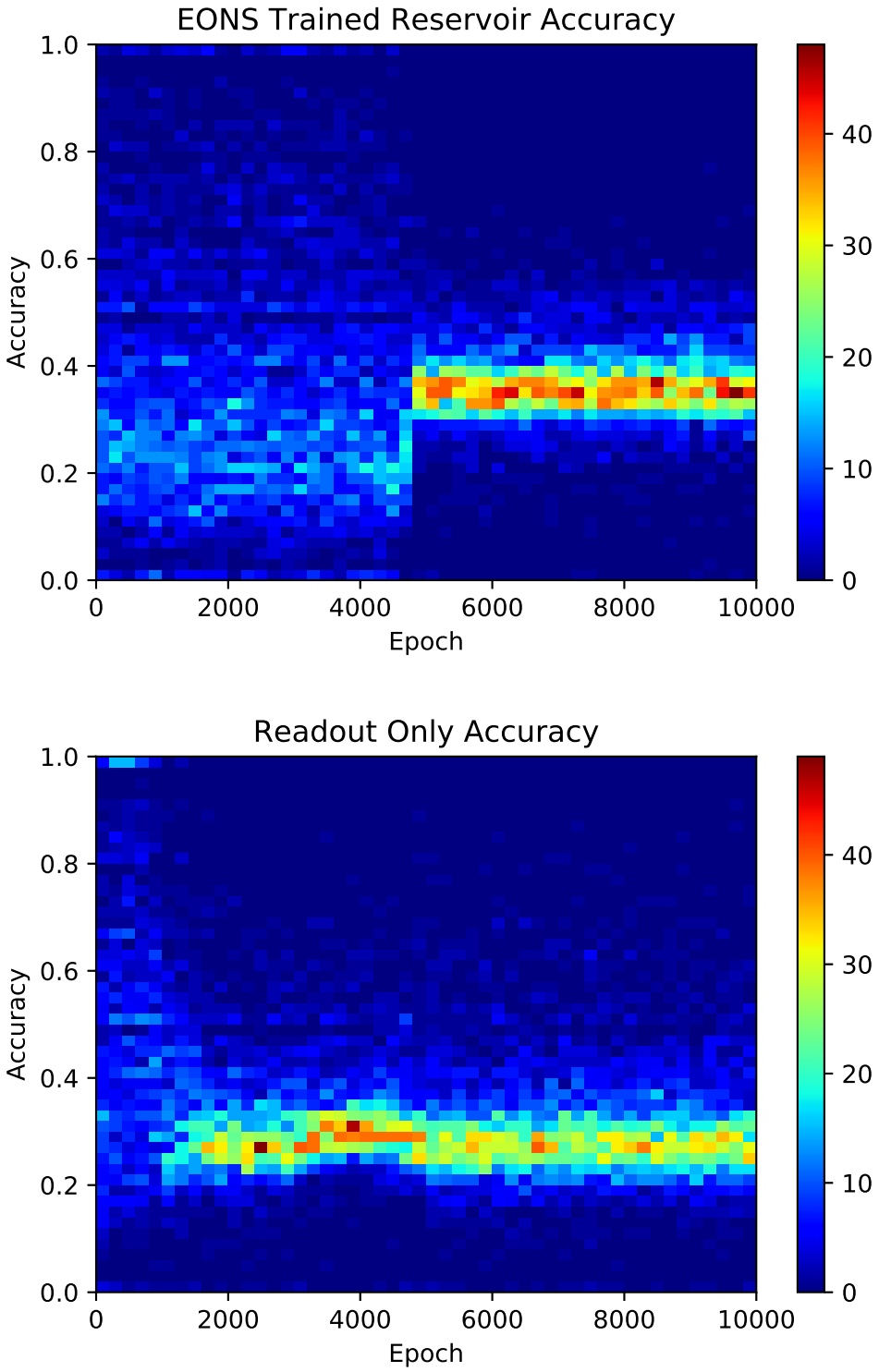
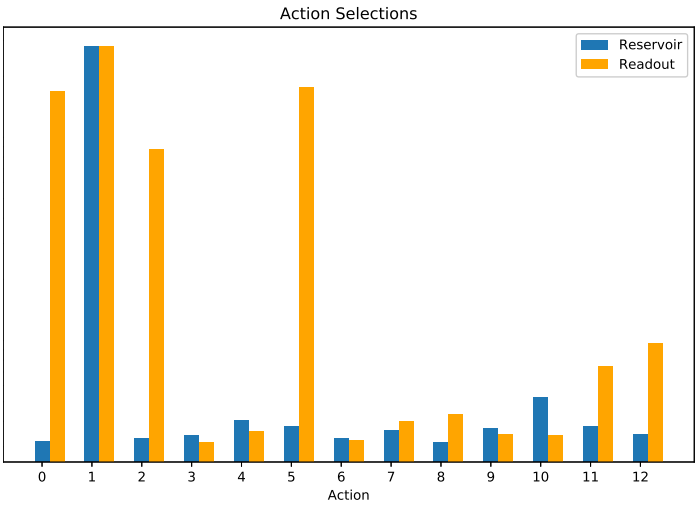


Figure 5.7: These heatmaps demonstrate the accuracy the best network throughout training time.



Action Table

0	Rotate CCW
1	No Action
2	Rotate CW
3	Fire
4	Thrust
5	Rotate CCW + Fire
6	Rotate CW + Fire
7	Rotate CCW + Thrust
8	Rotate CW + Thrust
9	Thrust + Fire
10	Rotate CCW + Thrust + Fire
11	Rotate CW + Thrust + Fire
12	No Thrust

Figure 5.8: Relative distribution of output events for Asteroids while training the readout.

The EONS trajectories can be seen in figure 5.9. The best network exists at epoch 0, and like Asteroids, a successful network leads to long simulation times, so that EONS run only completed one epoch.

Training the readout layer on the best reservoir is also depicted in figure 5.10. The temperature value combined with the reservoir provides a smooth learning curve. The readout layer alone is clearly overfitting and suffering from the problem of high variance gradients that are known to come with policy gradient methods. Therefore, the reservoir is a direct improvement in the case of Spaceinvaders.

5.6.3 Bowman

The Bowman fitness function is defined by $\frac{birds_shot^2}{birds_spawned \cdot arrows_shot}$. Like the others, it rewards both score and efficiency. In Figure 5.11, we show the EONS trajectories for Bowman. Unlike the training for Astersoids and Spaceinvaders, there was no one EONS run that vastly outperformed the others. For that reason, the training runs ran for similar numbers of epochs.

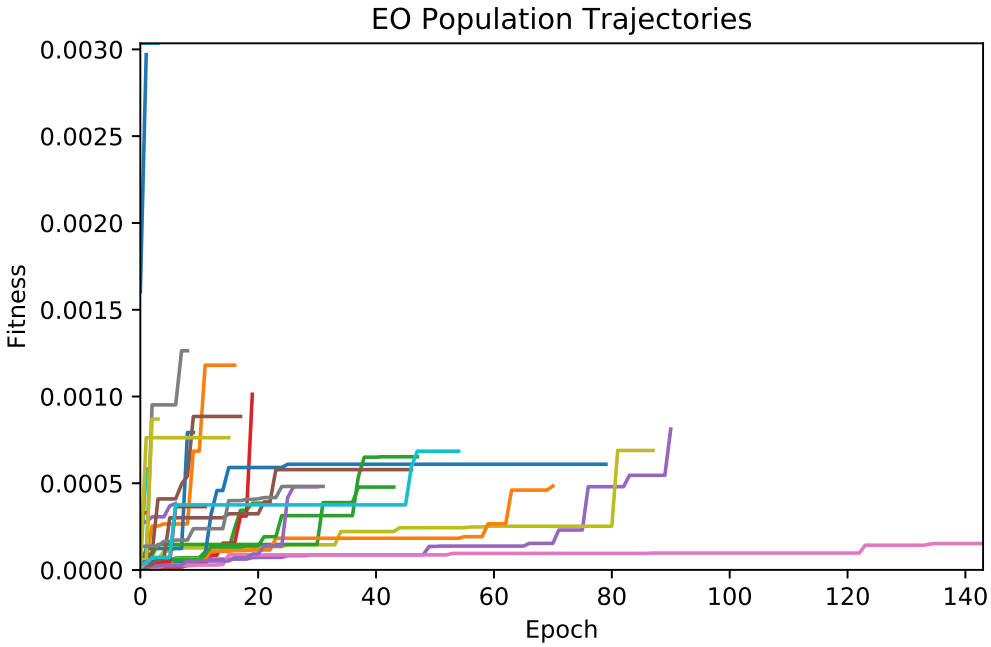


Figure 5.9: The EONS trajectories for Spaceinvaders.

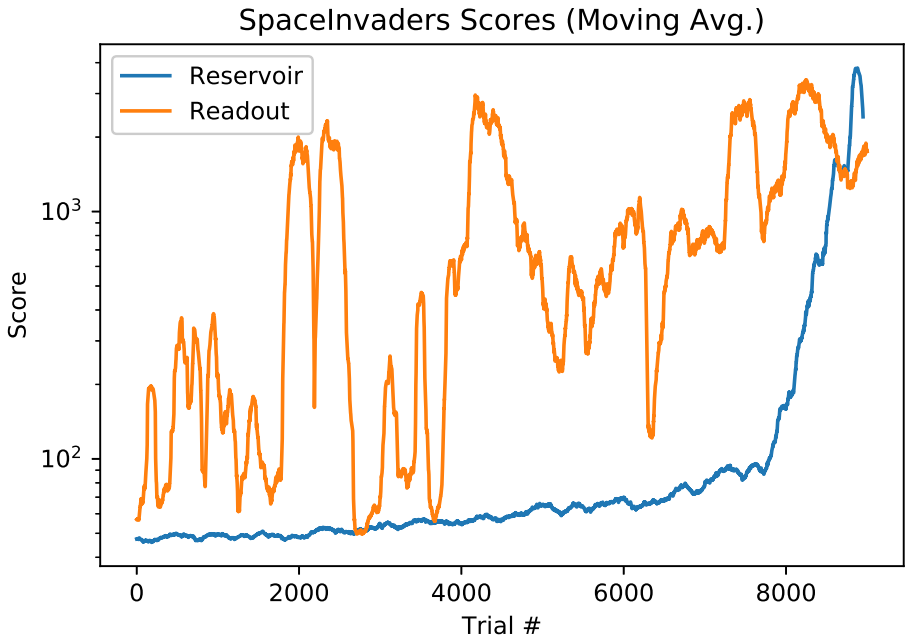


Figure 5.10: Training the readout layer for Spaceinvaders, both without a reservoir and with the best reservoir produced by EONS.

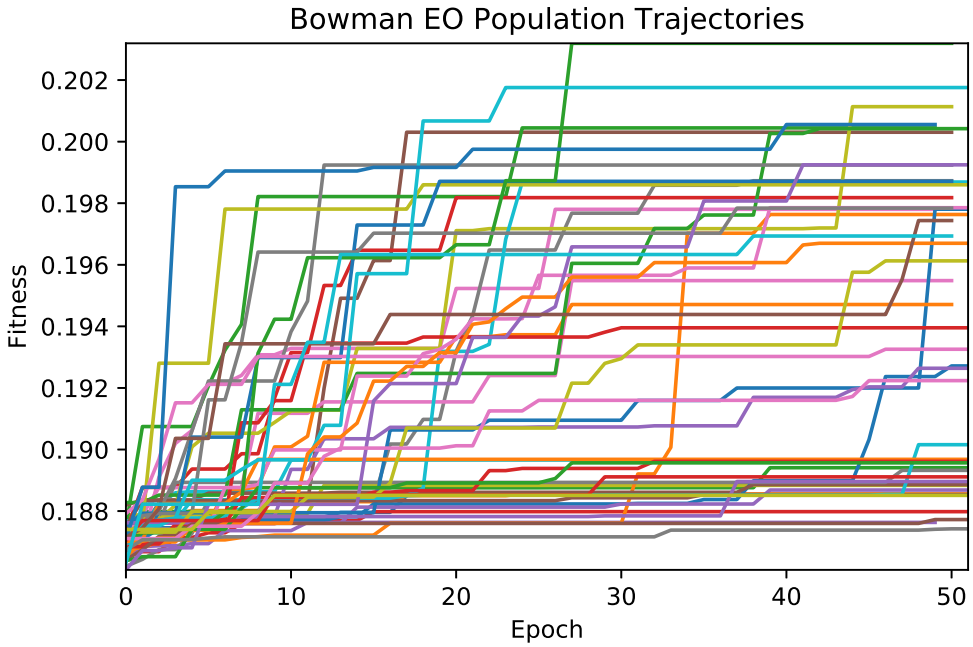


Figure 5.11: The EONS trajectories for Bowman.

In training the readouts, the non-reservoir training failed completely, locking onto bad behavior and not subsequently improving. In other words, the policy gradient fails to converge with the readout layer alone. On the other hand, the readout for the reservoir improved the score from the EONS run, and was vastly superior to the other readout. These results can be seen in figure 5.12. We can conclude that in this case, the reservoir provide a useful transformation to game state, so that the readout could be trained.

With Bowman, we can compare these results to results obtained by EONS optimizing networks with no readout [88]. In those results, the fitness scores on trained networks ranged from 0.35 to 0.94, with a median of 0.66. These results are signifcantly superior to the results here, indicating that more work needs to be done to make the combination of reservoir computing and reinforcement learning competitive.

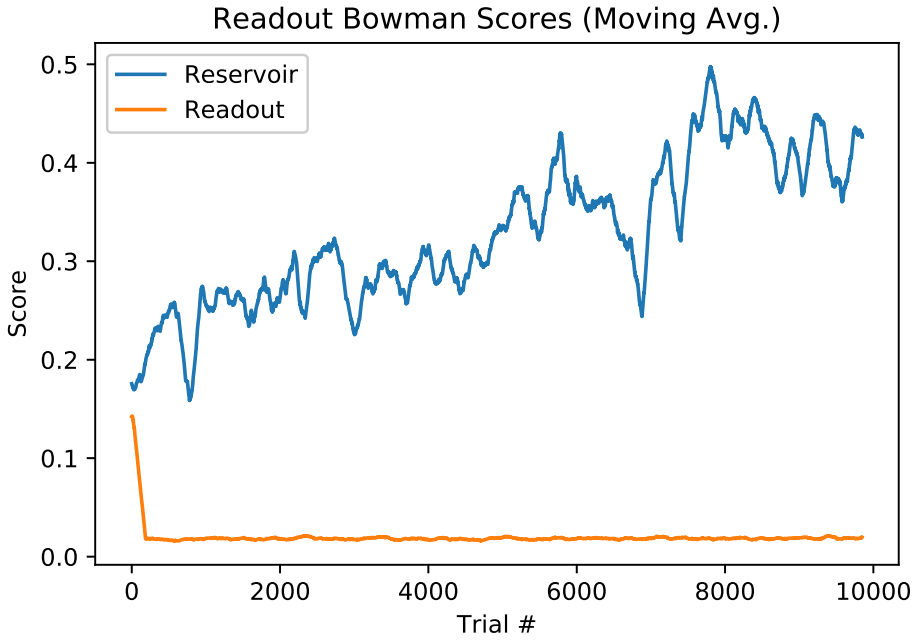


Figure 5.12: Training the readout layer for Bowman, both without a reservoir and with the best reservoir produced by EONS.

5.7 Discussion and Conclusion

The results of this experiment are varied. There are some promising data points, as in each case, the EONS-trained reservoir outperformed the readout trained without the reservoir. However, in the case of Bowman, where we have other results for comparison, the combination of reservoir computing and reinforcement learning is not competitive with using EONS and no readout. We conclude that while we have demonstrated promise with this approach, more research needs to be performed to make it competitive.

The most debilitating challenge that we currently face in this research area is the number of hyperparameters that must be set in any particular experiment. In our experiments, the following hyperparameters must be set, and each can impact the effectiveness of the final reservoir and readout:

- Reservoir size and synaptic density.

- The number of outputs from the reservoir.
- The gamma value (see Equation 5.2 above).
- The learning rate and learning rate schedule.
- Application simulation time.
- Duration of an episode with respect to the neuromorphic processor.
- Reward function.
- Temperature.
- Number of epochs for readout training.
- EONS population size.
- EONS number of epochs.
- EONS selection algorithm.
- EONS mutation and crossover rate.

The sheer number of hyperparameters hinders an effective search, which we surmise is why the results of this work are so varied. Until a more effective setting of the hyperparameters may be determined, work in this vein is likely to have the results that we have shown – varied successes and failures with little guidance on the reasons. Interestingly, a recent experiment by Ponghiran *et al* [92] reached the same conclusion. Their work focuses on the combination of liquid state machines and reinforcement learning on applications from OpenAI Gym. Rather than policy gradients, they used Q-Learning, but their results, like ours, demonstrated mixed successes without clear reasons for success or failure. Further work in this area clearly needs to address the setting of hyperparameters.

Chapter 6

Future Directions & Conclusions

Liquid State Machines are an interesting phenomenon in the world of spiking neural networks. Their theoretical capabilities are well understood; however, the major technical challenge in using liquid state machines is finding valid approaches to designing high performing reservoirs on practical applications. In chapter three, we compared the abilities of liquid state machines against EONS and deep learning in classification applications. We determined that LSMs have the ability to perform as well as the other machine learning techniques. Chapter four explored the conjunction of EONS and reservoir computing to determine if genetic algorithms were a reasonable approach for generating liquid state machines. The results demonstrated useful convergence properties for finding liquid state machines in an intelligent way versus performing a grid search. Finally, chapter six incorporated liquid state machines with reinforcement learning. Policy gradients were used to train the readout layer for control applications. The readout layer alone was compared to liquid state machine performance. The results were varied, but they contain direct examples of liquid state machines improving the behavior of the vanilla policy gradient implementation.

Hyperparameters continue to be a hindrance to performance. Finding well-performing reservoirs generally relies on heuristic approaches, as direct construction fails more often than one would like for practical applications. There are numerous future directions to explore in liquid state machines. Deep liquid state machines are

a particularly exciting front [117, 136] that incorporate preprocessing of features as well as attention mechanisms for focusing on specific elements of reservoir activity. One instance of a deep liquid state machine has been shown to achieve state of the art results on a video recognition task when combined with a traditional deep neural network for feature extraction. The only other paper discussing liquid state machines and reinforcement learning on Atari games (to the dissertation author's knowledge) was published on arxiv during the last few months of writing this dissertation [92]. The authors of that paper explore liquid state machines for reinforcement learning using q-learning, which serves as a complement for the policy gradient method explored in this dissertation. Their results, like ours, are inconclusive, suffering from the high number of hyperparameters that must be set to perform an experiment.

One of the critical issues in this dissertation is that of genetic algorithms (both with and without reinforcement learning) for generating liquid state machines taking an extended period of time to train. Expanding on existing genetic algorithm efficiency or finding other heuristics for generating valid and applicable liquid state machines is a promising topic for extended study.

To conclude, reservoir computing is an attractive paradigm for neuromorphic computing, especially when considering the attractive power savings these systems offer combined with the demonstrated robustness of reservoirs. Like other work, this dissertation provides data points that support the further exploration of reservoir computing. The two major challenges that must be addressed are embodied in the following two questions:

1. How do we systematically generate good reservoirs, both in general and for specific problems?

2. How de we effectively set the myriad hyperparameters involved with reservoir computing in general, with with the combination of reservoir computing and reinforcement learning in particular.

The successful application of reservoir computing to a variety of problems will be based on whether these questions may be answered adequately.

Bibliography

[1] Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J., Devin, M., Ghemawat, S., Irving, G., Isard, M., et al. (2016). Tensorflow: A system for large-scale machine learning. In *OSDI*, volume 16, pages 265–283. 4, 22, 27

[2] Akopyan, F., Sawada, J., Cassidy, A., Alvarez-Icaza, R., Arthur, J., Merolla, P., Imam, N., Nakamura, Y., Datta, P., Nam, G.-J., et al. (2015). Truenorth: Design and tool flow of a 65 mw 1 million neuron programmable neurosynaptic chip. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 34(10):1537–1557. 19

[3] Alba, E. and Chicano, J. F. (2004). Training neural networks with ga hybrid algorithms. In *Genetic and Evolutionary Computation Conference*, pages 852–863. Springer. 19

[4] Alexandre, L. A., Embrechts, M. J., and Linton, J. (2009). Benchmarking reservoir computing on time-independent classification tasks. In *Neural Networks, 2009. IJCNN 2009. International Joint Conference on*, pages 89–93. IEEE. 20, 21, 32, 43, 44

[5] Andrzejak, R. G., Lehnertz, K., Mormann, F., Rieke, C., David, P., and Elger, C. E. (2001). Indications of nonlinear deterministic and finite-dimensional structures in time series of brain electrical activity: Dependence on recording region and brain state. *Physical Review E*, 64(6):061907. 34

[6] Angeline, P. J., Saunders, G. M., and Pollack, J. B. (1994). An evolutionary algorithm that constructs recurrent neural networks. *IEEE transactions on Neural Networks*, 5(1):54–65. 22

[7] Antonelo, E. A. and Schrauwen, B. (2014). On learning navigation behaviors for small mobile robots with reservoir computing architectures. *IEEE transactions on neural networks and learning systems*, 26(4):763–780. 23, 58

[8] Appeltant, L., Soriano, M. C., Van der Sande, G., Danckaert, J., Massar, S., Dambre, J., Schrauwen, B., Mirasso, C. R., and Fischer, I. (2011).

Information processing using a single dynamical node as complex system. *Nature Communications*, 2. 31

[9] Benjamin, B. V., Gao, P., McQuinn, E., Choudhary, S., Chandrasekaran, A. R., Bussat, J.-M., Alvarez-Icaza, R., Arthur, J. V., Merolla, P. A., and Boahen, K. (2014). Neurogrid: A mixed-analog-digital multichip system for large-scale neural simulations. *Proceedings of the IEEE*, 102(5):699–716. 19

[10] Bohte, S. M., Kok, J. N., and La Poutre, H. (2002). Error-backpropagation in temporally encoded networks of spiking neurons. *Neurocomputing*, 48(1):17–37. 19

[11] Burgsteiner, H. (2019). On learning with recurrent spiking neural networks and their applications to robot control with real-world devices. 20

[12] Burgsteiner, H., Kröll, M., Leopold, A., and Steinbauer, G. (2007). Movement prediction from real-world images using a liquid state machine. *Applied Intelligence*, 26(2):99–109. 20

[13] Burkitt, A. N. (2006). A review of the integrate-and-fire neuron model: I. homogeneous synaptic input. *Biological cybernetics*, 95(1):1–19. 5

[14] Carlson, K. D., Dutt, N., Nageswaran, J. M., and Krichmar, J. L. (2013). Design space exploration and parameter tuning for neuromorphic applications. In *Proceedings of the Ninth IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*, page 20. IEEE Press. 19

[15] Chakma, G., Adnan, M. M., Wyer, A. R., Weiss, R., Schuman, C. D., and Rose, G. S. (2018). Memristive mixed-signal neuromorphic systems: Energy-efficient learning at the circuit-level. *IEEE Journal on Emerging and Selected topics in Circuits and Systems (JETCAS)*, 8(1):125–136. 8

[16] Chakma, G., Sayyaparaju, S., Weiss, R., and Rose, G. S. (2017). A mixed-signal approach to memristive neuromorphic system design. In *60th IEEE International Midwest Symposium on Circuits and Systems (MWSCAS)*, Boston, MA. 8

[17] Chella, A. and Rizzo, R. (2005). Time-varying signals classification using a liquid state machine. In *Biological and Artificial Intelligence Environments*, pages 133–139. Springer. 20

[18] Chiu, C.-C., Sainath, T. N., Wu, Y., Prabhavalkar, R., Nguyen, P., Chen, Z., Kannan, A., Weiss, R. J., Rao, K., Gonina, E., et al. (2018). State-of-the-art speech recognition with sequence-to-sequence models. In *2018 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 4774–4778. IEEE. 4

[19] Chollet, F. et al. (2015). Keras. https://keras.io. 22, 27

[20] Cole, R. A., Yan, Y., Mak, B., Fanty, M., and Bailey, T. (1996). The contribution of consonants versus vowels to word recognition in fluent speech. In *Acoustics, Speech, and Signal Processing, 1996. ICASSP-96. Conference Proceedings., 1996 IEEE International Conference on*, volume 2, pages 853–856. IEEE. 34

[21] Davies, M., Srinivasa, N., Lin, T.-H., Chinya, G., Cao, Y., Choday, S. H., Dimou, G., Joshi, P., Imam, N., Jain, S., et al. (2018). Loihi: A neuromorphic manycore processor with on-chip learning. *IEEE Micro*, 38(1):82–99. 6, 19

[22] Davis, S. B. and Mermelstein, P. (1980). Comparison of parametric representations for monosyllabic word recognition in continuously spoken sentences. *IEEE Transactions on Acoustics, Speech, and Signal Processing*, 28(4):357–366. 34

[23] Dean, M. E., Chan, J., Daffron, C., Disney, A., Reynolds, J., Rose, G. S., Plank, J. S., Birdwell, J., and Schuman, C. D. (2016). An application development platform for neuromorphic computing. In *International Joint Conference on Neural Networks*, Vancouver. 9, 14, 22

[24] DeepSig, Inc (2017). Resources for radio machine learning. https://www.deepsig.io/resources. 33

[25] Du, C., Cai, F., Zidan, M. A., Ma, W., Lee, S. H., and Lu, W. D. (2017). Reservoir computing using dynamic memristors for temporal information processing. *Nature communications*, 8(1):2204. 20

[26] Fernando, C. and Sojakka, S. (2003). Pattern recognition in a bucket. In *European Conference on Artificial Life*, pages 588–597. Springer. 21

[27] Ferreira, A. A. and Ludermir, T. B. (2009). Genetic algorithm for reservoir computing optimization. In *Neural Networks, 2009. IJCNN 2009. International Joint Conference on*, pages 811–815. IEEE. 11, 31

[28] Fisher, R. A. (1936a). The use of multiple measurements in taxonomic problems. *Annals of eugenics*, 7(2):179–188. ix, 14, 15

[29] Fisher, R. A. (1936b). The use of multiple measurements in taxonomic problems. *Annual Eugenics*, 7-II:179–188. 32

[30] Floreano, D., Dürr, P., and Mattiussi, C. (2008). Neuroevolution: from architectures to learning. *Evolutionary Intelligence*, 1(1):47–62. 19

[31] Fujii, K. and Nakajima, K. (2016). Harnessing disordered quantum dynamics for machine learning. *arXiv preprint arXiv:1602.08159*. 21

[32] Gao, B. and Pavel, L. (2017). On the properties of the softmax function with application in game theory and reinforcement learning. 62

[33] García-Pedrajas, N., Ortiz-Boyer, D., and Hervás-Martínez, C. (2006). An alternative approach for neural network evolution with a genetic algorithm: Crossover by combinatorial optimization. *Neural Networks*, 19(4):514–528. 22

[34] Ghosh-Dastidar, S. and Adeli, H. (2009). Spiking neural networks. *International journal of neural systems*, 19(04):295–308. 5

[35] Graves, A., Mohamed, A., and Hinton, G. E. (2013). Speech recognition with deep recurrent neural networks. *CoRR*, abs/1303.5778. 27

[36] Greff, K., Srivastava, R. K., Koutník, J., Steunebrink, B. R., and Schmidhuber, J. (2016). Lstm: A search space odyssey. *IEEE transactions on neural networks and learning systems*, 28(10):2222–2232. 4

[37] Happel, B. L. and Murre, J. M. (1994). Design and evolution of modular neural network architectures. *Neural networks*, 7(6-7):985–1004. 22

[38] Hastie, T., Tibshirani, R., and Friedman, J. (2009). *The Elements of Statistical Learning: Data Mining, Inference, and Prediction*. Springer, New York, NY. 27

[39] He, K., Zhang, X., Ren, S., and Sun, J. (2016). Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778. 4

[40] He, Y.-L., Zhang, X.-L., Ao, W., and Huang, J. Z. (2018). Determining the optimal temperature parameter for softmax function in reinforcement learning. *Applied Soft Computing*, 70:80 – 85. 62

[41] Hebb, D. O. (2005). *The organization of behavior: A neuropsychological theory*. Psychology Press. 3

[42] Hecht-Nielsen, R. (1992). Theory of the backpropagation neural network. In *Neural networks for perception*, pages 65–93. Elsevier. 3

[43] Hochreiter, S. and Schmidhuber, J. (1997). Long short-term memory. *Neural Computation*, 9(8):1735–1780. 27

[44] Hourdakis, E. and Trahanias, P. (2011). Improving the classification performance of liquid state machines based on the separation property. In Iliadis, L. and Jayne, C., editors, *Engineering Applications of Neural Networks*, pages 52–62, Berlin, Heidelberg. Springer Berlin Heidelberg. 43

[45] Jaeger, H. (2001a). The "echo state" approach to analysing and training recurrent neural networks. GMD Report 148, German National

Research Center for Information Technology. Reprint in 2010 with erratum note at http://minds.jacobs-university.de/sites/default/files/uploads/papers/EchoStatesTechRep.pdf. 30, 31, 40

[46] Jaeger, H. (2001b). The echo state approach to analysing and training recurrent neural networks-with an erratum note. *Bonn, Germany: German National Research Center for Information Technology GMD Technical Report*, 148(34):13. 10, 20, 21

[47] Jaeger, H. (2002). Short term memory in echo state networks. 20

[48] Jaeger, H., Lukoševičius, M., Popovici, D., and Siewert, U. (2007). Optimization and applications of echo state networks with leaky-integrator neurons. *Neural networks*, 20(3):335–352. 21

[49] Jain, A. K., Jianchang Mao, and Mohiuddin, K. M. (1996). Artificial neural networks: a tutorial. *Computer*, 29(3):31–44. 2

[50] Jia, Y., Shelhamer, E., Donahue, J., Karayev, S., Long, J., Girshick, R., Guadarrama, S., and Darrell, T. (2014). Caffe: Convolutional architecture for fast feature embedding. *arXiv preprint arXiv:1408.5093*. 22

[51] Kaelbling, L. P., Littman, M. L., and Moore, A. W. (1996). Reinforcement learning: A survey. *Journal of artificial intelligence research*, 4:237–285. 23, 58

[52] Kasabov, N., Feigin, V., Hou, Z.-G., Chen, Y., Liang, L., Krishnamurthi, R., Othman, M., and Parmar, P. (2014). Evolving spiking neural networks for personalised modelling, classification and prediction of spatio-temporal patterns with a case study on stroke. *Neurocomputing*, 134:269–279. 19

[53] Konda, V. R. and Tsitsiklis, J. N. (2000). Actor-critic algorithms. In *Advances in neural information processing systems*, pages 1008–1014. 23

[54] Koza, J. R. (1997). Genetic programming. 22

[55] Kudithipudi, D., Saleh, Q., Merkel, C., Thesing, J., and Wysocki, B. (2016). Design and analysis of a neuromemristive reservoir computing architecture for biosignal processing. *Frontiers in neuroscience*, 9:502. 20

[56] Kukačka, J., Golkov, V., and Cremers, D. (2017). Regularization for deep learning: A taxonomy. *arXiv preprint arXiv:1710.10686*. 71

[57] Kulkarni, M. S. and Teuscher, C. (2012). Memristor-based reservoir computing. In *Nanoscale Architectures (NANOARCH), 2012 IEEE/ACM International Symposium on*, pages 226–232. IEEE. 20

[58] LeCun, Y., Bengio, Y., and Hinton, G. (2015). Deep learning. *nature*, 521(7553):436. 3

[59] LeCun, Y., Bottou, L., Bengio, Y., and Haffner, P. (1998). Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278 – 2324. 27

[60] Lenz, I., Lee, H., and Saxena, A. (2015). Deep learning for detecting robotic grasps. *The International Journal of Robotics Research*, 34(4-5):705–724. 4

[61] Lichman, M. (2013). UCI machine learning repository. 32

[62] Liu, W., Anguelov, D., Erhan, D., Szegedy, C., Reed, S., Fu, C.-Y., and Berg, A. C. (2016). Ssd: Single shot multibox detector. In *European conference on computer vision*, pages 21–37. Springer. 4

[63] Lukoševičius, M. (2012). A practical guide to applying echo state networks. In *Neural networks: Tricks of the trade*, pages 659–686. Springer. 21

[64] Lukoševičius, M. and Jaeger, H. (2009). Reservoir computing approaches to recurrent neural network training. *Computer Science Review*, 3(3):127–149. 10, 21

[65] Lukoševičius, M., Jaeger, H., and Schrauwen, B. (2012a). Reservoir computing trends. *KI - Künstliche Intelligenz*, 26(4):365–371. 20, 22, 45, 48

[66] Lukoševičius, M., Jaeger, H., and Schrauwen, B. (2012b). Reservoir computing trends. *KI-Künstliche Intelligenz*, 26(4):365–371. 20, 21

[67] Maass, W. (1997). Networks of spiking neurons: the third generation of neural network models. *Neural networks*, 10(9):1659–1671. 5, 18

[68] Maass, W. and Markram, H. (2004). On the computational power of circuits of spiking neurons. *Journal of computer and system sciences*, 69(4):593–616. 11

[69] Maass, W., Natschläger, T., and Markram, H. (2002a). Real-time computing without stable states: A new framework for neural computation based on perturbations. *Neural computation*, 14(11):2531–2560. 10, 20, 22

[70] Maass, W., Natschläger, T., and Markram, H. (2002b). Real-time computing without stable states: A new framework for neural computation based on perturbations. *Neural computation*, 14(11):2531–2560. 30, 31

[71] Markram, H. (2012). The human brain project. *Scientific American*, 306(6):50–55. 19

[72] Masquelier, T. and Thorpe, S. J. (2007). Unsupervised learning of visual features through spike timing dependent plasticity. *PLoS computational biology*, 3(2):e31. 19

[73] McCulloch, W. S. and Pitts, W. (1943). A logical calculus of the ideas immanent in nervous activity. *The bulletin of mathematical biophysics*, 5(4):115–133. 3

[74] Mead, C. (1990). Neuromorphic electronic systems. *Proceedings of the IEEE*, 78(10):1629–1636. 1

[75] Minsky, M. L. and Papert, S. A. (1988). Perceptrons: expanded edition. 3

[76] Mitchell, J. P. (2018). Danna2: Dynamic adaptive neural networks arrays. Master's thesis. 7

[77] Mitchell, J. P., Bruer, G., Dean, M. E., Plank, J. S., Rose, G. S., and Schuman, C. D. (2017). NeoN: Neuromorphic control for autonomous robotic navigation. In *IEEE 5th International Symposium on Robotics and Intelligent Sensors*, pages 136–142, Ottawa, Canada. 30

[78] Mitchell, J. P., Dean, M. E., Bruer, G., Plank, J. S., and Rose, G. S. (2018). DANNA 2: Dynamic adaptive neural network arrays. In *International Conference on Neuromorphic Computing Systems*, Knoxville, TN. ACM. 42

[79] Mitchell, M. (1998). *An Introduction to Genetic Algorithms*. MIT Press, Cambridge, MA, USA. 8

[80] Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D., and Riedmiller, M. (2013). Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*. 23, 58, 63, 66

[81] Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., Graves, A., Riedmiller, M., Fidjeland, A. K., Ostrovski, G., et al. (2015). Human-level control through deep reinforcement learning. *Nature*, 518(7540):529. 23, 58

[82] Norton, D. and Ventura, D. (2010). Improving liquid state machines through iterative refinement of the reservoir. *Neurocomputing*, 73:2893–2904. 34, 36

[83] O'Shea, T. J., Corgan, J., and Clancy, T. C. (2016). Convolutional radio modulation recognition networks. arXiv:1602.04105v3. 33, 36, 37

[84] Palumbo, F., Gallicchio, C., Pucci, R., and Micheli, A. (2016). Human activity recognition using multisensor data fusion based on reservoir computing. *Journal of Ambient Intelligence and Smart Environments*, 8(2):87–107. 21

[85] Paquot, Y., Duport, F., Smerieri, A., Dambre, J., Schrauwen, B., Haelterman, M., and Massar, S. (2012). Optoelectronic reservoir computing. *Scientific reports*, 2. 21

[86] Paszke, A., Gross, S., Chintala, S., Chanan, G., Yang, E., DeVito, Z., Lin, Z., Desmaison, A., Antiga, L., and Lerer, A. (2017). Automatic differentiation in pytorch. 4, 22

[87] Pavlidis, N., Tasoulis, O., Plagianakos, V. P., Nikiforidis, G., and Vrahatis, M. (2005). Spiking neural network training using evolutionary algorithms. In *Neural Networks, 2005. IJCNN'05. Proceedings. 2005 IEEE International Joint Conference on*, volume 4, pages 2190–2194. IEEE. 19

[88] Plank, J. S., Rizzo, C., Shahat, K., Bruer, G., Dixon, T., Goin, M., Zhao, G., Anantharaj, J., Schuman, C. D., Dean, M. E., Rose, G. S., Cady, N. C., and Van Nostrand, J. (2019). The TENNLab suite of LIDAR-based control applications for recurrent, spiking, neuromorphic systems. In *44th Annual GOMACTech Conference*, Albuquerque. 14, 63, 75

[89] Plank, J. S., Rose, G. S., Dean, M. E., Schuman, C. D., and Cady, N. C. (2017). A unified hardware/software co-design framework for neuromorphic computing devices and applications. In *IEEE International Conference on Rebooting Computing (ICRC 2017)*, Washington, DC. 30

[90] Plank, J. S., Schuman, C. D., Bruer, G., Dean, M. E., and Rose, G. S. (2018). The TENNLab exploratory neuromorphic computing framework. *IEEE Letters of the Computer Society*, 1(2):17–20. 2, 14, 42

[91] Polepalli, A., Soures, N., and Kudithipudi, D. (2016). Reconfigurable digital design of a liquid state machine for spatio-temporal data. In *Proceedings of the 3rd ACM International Conference on Nanoscale Computing and Communication*, page 15. ACM. 5

[92] Ponghiran, W., Srinivasan, G., and Roy, K. (2019). Reinforcement learning with low-complexity liquid state machines. *CoRR*, abs/1906.01695. 23, 58, 77, 79

[93] Rast, A. D., Jin, X., Galluppi, F., Plana, L. A., Patterson, C., and Furber, S. (2010). Scalable event-driven native parallel processing: the spinnaker

neuromimetic system. In *Proceedings of the 7th ACM international conference on Computing frontiers*, pages 21–30. ACM. 19

[94] Reverdy, P. and Leonard, N. E. (2015). Parameter estimation in softmax decision-making models with linear objective functions. *IEEE Transactions on Automation Science and Engineering*, 13(1):54–67. 62

[95] Reynolds, J. J. M., Plank, J. S., and Schuman, C. D. (2019). Intelligent reservoir generation for liquid state machines using evolutionary optimization. In *IJCNN: The International Joint Conference on Neural Networks*, Budapest. 41

[96] Reynolds, J. J. M., Plank, J. S., Schuman, C. D., Bruer, G., Disney, A. W., Dean, M., and Rose, G. S. (2018). A comparison of neuromorphic classification tasks. In *International Conference on Neuromorphic Computing Systems*, Knoxville, TN. ACM. 24

[97] Ronneberger, O., Fischer, P., and Brox, T. (2015). U-net: Convolutional networks for biomedical image segmentation. In *International Conference on Medical image computing and computer-assisted intervention*, pages 234–241. Springer. 4

[98] Rosenblatt, F. (1958). The perceptron: a probabilistic model for information storage and organization in the brain. *Psychological review*, 65(6):386. 3

[99] Rousseeuw, P. J. (1987). Silhouettes: a graphical aid to the interpretation and validation of cluster analysis. *Journal of computational and applied mathematics*, 20:53–65. 45

[100] Ruder, S. (2016). An overview of gradient descent optimization algorithms. *arXiv preprint arXiv:1609.04747*. 3

[101] Schrauwen, B., D'Haene, M., Verstraeten, D., and Van Campenhout, J. (2007). Compact hardware for real-time speech recognition using a liquid state machine. In *2007 International Joint Conference on Neural Networks*, pages 1097–1102. IEEE. 20

[102] Schuman, C. D. (2015). *Neuroscience-Inspired Dynamic Architectures*. PhD thesis, University of Tennessee. 6, 7, 66

[103] Schuman, C. D. (2017). Poster: The effect of biologically-inspired mechanisms in spiking neural networks for neuromorphic implementation. 5th Neuro Inspired Computational Elements Workshop (NICE 2017). 28

[104] Schuman, C. D., Birdwell, J. D., and Dean, M. E. (2014a). Neuroscience-inspired dynamic architectures. In *Biomedical Science and Engineering Center Conference (BSEC), Oak Ridge National Laboratory*. 28

[105] Schuman, C. D., Birdwell, J. D., and Dean, M. E. (2014b). Spatiotemporal classification using neuroscience-inspired dynamic architectures. *Procedia Computer Science*, 41:89–97. 28

[106] Schuman, C. D., Disney, A., Singh, S. P., Bruer, G., Mitchell, J. P., Klibisz, A., and Plank, J. S. (2016a). Parallel evolutionary optimization for neuromorphic network training. In *Machine Learning in HPC Environments, Supercomputing 2016*, Salt Lake City. 22, 30

[107] Schuman, C. D., Plank, J. S., Bruer, G., and Anantharaj, J. (2019). Non-traditional input encoding schemes for spiking neuromorphic systems. In *2019 International Joint Conference on Neural Networks (IJCNN)*, pages 1–10. 16

[108] Schuman, C. D., Plank, J. S., Disney, A., and Reynolds, J. (2016b). An evolutionary optimization framework for neural networks and neuromorphic architectures. In *International Joint Conference on Neural Networks*, Vancouver. 2, 9, 14, 19, 22, 28, 29, 30, 36, 42

[109] Schuman, C. D., Potok, T. E., Patton, R. M., Birdwell, J. D., Dean, M. E., Rose, G. S., and Plank, J. S. (2017a). A survey of neuromorphic computing and neural networks in hardware. *arXiv preprint arXiv:1705.06963*. 18

[110] Schuman, C. D., Potok, T. E., Patton, R. M., Birdwell, J. D., Dean, M. E., Rose, G. S., and Plank, J. S. (2017b). A survey of neuromorphic computing and neural networks in hardware. arXiv:1705.06963. 19

[111] Seide, F. and Agarwal, A. (2016). Cntk: Microsoft's open-source deep-learning toolkit. In *Proceedings of the 22Nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '16, pages 2135–2135, New York, NY, USA. ACM. 22, 27

[112] Severa, W., Vineyard, C. M., Dellana, R., Verzi, S. J., and Aimone, J. B. (2019). Training deep neural networks for binary communication with the whetstone method. *Nature Machine Intelligence*, 1(2):8694. 19

[113] Siegelmann, H. T. and Sontag, E. D. (1992). On the computational power of neural nets. In *Proceedings of the fifth annual workshop on Computational learning theory*, pages 440–449. ACM. 5

[114] Sigillito, V. G., Wing, S. P., Hutton, L. V., and Baker, K. B. (1989). Classification of radar returns from the ionosphere using neural networks. *Johns Hopkins APL Tech. Dig*, vol. 10:262–266. in. 43

[115] Silver, D., Huang, A., Maddison, C. J., Guez, A., Sifre, L., van den Driessche, G., Schrittwieser, J., Antonoglou, I., Panneershelvam, V., Lanctot, M., Dieleman, S., Grewe, D., Nham, J., Kalchbrenner, N., Sutskever, I., Lillicrap, T., Leach, M., Kavukcuoglu, K., Graepel, T., and Hassabis, D. (2016). Mastering the game of go with deep neural networks and tree search. *Nature*, 529:484–503. 23

[116] Soures, N., Hays, L., and Kudithipudi, D. (2017). Robustness of a memristor based liquid state machine. In *Neural Networks (IJCNN), 2017 International Joint Conference on*, pages 2414–2420. IEEE. 20, 34

[117] Soures, N. and Kudithipudi, D. (2019). Deep liquid state machines with neural plasticity for video activity recognition. *Frontiers in Neuroscience*, 13:686. 79

[118] Stanley, K. O. and Miikkulainen, R. (2002a). Evolving neural networks through augmenting topologies. *Evol. Comput.*, 10(2):99–127. 8

[119] Stanley, K. O. and Miikkulainen, R. (2002b). Evolving neural networks through augmenting topologies. *Evolutionary computation*, 10(2):99–127. 19, 22

[120] Stimberg, M., Brette, R., and Goodman, D. F. (2019). Brian 2, an intuitive and efficient neural simulator. *eLife*, 8:e47314. 6

[121] Street, W. N., Wolberg, W. H., and Mangasarian, O. (1993). Nuclear feature extraction for breast tumor diagnosis. In *IS&T/SPIE International Symposium on Electronic Imaging: Science and Technology*, volume 1 905, pages 861–870, San Jose, CA. 32

[122] Subasi, A. and Gursoy, M. I. (2010). Eeg signal classification using pca, ica, lda and support vector machines. *Expert Systems with Applications*, 37(12):8659–8666. 34, 36

[123] Sutton, R. S., Barto, A. G., et al. (1998). *Introduction to reinforcement learning*, volume 2. MIT press Cambridge. 23, 58, 59, 60, 62

[124] Sutton, R. S., McAllester, D. A., Singh, S. P., and Mansour, Y. (2000). Policy gradient methods for reinforcement learning with function approximation. In *Advances in neural information processing systems*, pages 1057–1063. 23

[125] Tanaka, G., Yamane, T., Héroux, J. B., Nakane, R., Kanazawa, N., Takeda, S., Numata, H., Nakano, D., and Hirose, A. (2019). Recent advances in physical reservoir computing: A review. *Neural Networks*, 115:100 – 123. 11

[126] Tax, N., Verenich, I., La Rosa, M., and Dumas, M. (2017). Predictive business process monitoring with lstm neural networks. In *International Conference on Advanced Information Systems Engineering*, pages 477–492. Springer. 4

[127] Theano Development Team (2016). Theano: A Python framework for fast computation of mathematical expressions. *arXiv e-prints*, abs/1605.02688. 22, 27

[128] Van Hasselt, H., Guez, A., and Silver, D. (2016). Deep reinforcement learning with double q-learning. In *Thirtieth AAAI conference on artificial intelligence*. 23

[129] Van Veldhuizen, D. A. and Lamont, G. B. (2000). On measuring multiobjective evolutionary algorithm performance. In *Proceedings of the 2000 Congress on Evolutionary Computation. CEC00 (Cat. No. 00TH8512)*, volume 1, pages 204–211. IEEE. 22

[130] Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, Ł., and Polosukhin, I. (2017). Attention is all you need. In *Advances in neural information processing systems*, pages 5998–6008. 4

[131] Verstraeten, D. and Schrauwen, B. (2009). On the quantification of dynamics in reservoir computing. In *International Conference on Artificial Neural Networks*, pages 985–994. Springer. 11

[132] Verstraeten, D., Schrauwen, B., dHaene, M., and Stroobandt, D. (2007). An experimental unification of reservoir computing methods. *Neural networks*, 20(3):391–403. 2, 9

[133] Verstraeten, D., Schrauwen, B., and Stroobandt, D. (2005). Isolated word recognition using a liquid state machine. In *ESANN*, volume 5, pages 435–440. Citeseer. 20

[134] Vreeken, J. (2003). Spiking neural networks, an introduction. 5

[135] Waldrop, M. M. (2016). The chips are down for moore's law. *Nature*, 530(7589):144–147. 1

[136] Wang, Q. and Li, P. (2016). D-lsm: Deep liquid state machine with unsupervised recurrent reservoir tuning. In *2016 23rd International Conference on Pattern Recognition (ICPR)*, pages 2652–2657. 79

[137] Watkins, C. J. and Dayan, P. (1992). Q-learning. *Machine learning*, 8(3-4):279–292. 23

[138] Whitley, D. (1994). A genetic algorithm tutorial. *Statistics and computing*, 4(2):65–85. 22

[139] Williams, R. J. (1992). Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine learning*, 8(3-4):229–256. 62

[140] Wojcik, G. M. and Kaminski, W. A. (2004). Liquid state machine built of hodgkin–huxley neurons and pattern recognition. *Neurocomputing*, 58:245–251. 20

[141] Young, S. R., Rose, D. C., Karnowski, T. P., Lim, S.-H., and Patton, R. M. (2015). Optimizing deep learning hyper-parameters through an evolutionary algorithm. In *Proceedings of the Workshop on Machine Learning in High-Performance Computing Environments*, page 4. ACM. 22

[142] Zheng, F., Zhang, G., and Song, Z. (2001). Comparison of different implementations of mfcc. *Journal of Computer Science and Technology*, 16(6):582–589. 34

[143] Zitzler, E. and Thiele, L. (1998). An evolutionary algorithm for multiobjective optimization: The strength pareto approach. *TIK-report*, 43. 22

Appendix

A Readout Layer

This section contains a reproduction of the code used to generate results throughout this work. It's a multinomial logistic regression / single layer neural network with no hidden neurons. It contains implementations of a simple backpropagation, reinforcement learning, gradient checking, softmax, and other linear algorithms necessary for use throughout (e.g. matrix multiplication).

A.1 Readout Class

This is the overarching readout class. It contains the required vectors and variables, as well as functions. This subsection simply depicts the base class, and following sections will cover the rest of the functions.

```
#include <cmath>
#include <vector>
#include <random>
#include <cstdio>
#include <algorithm>
#include <cfloat>
#include <cstdlib>
#include <fstream>
#include <utility>
#include <sstream>
```

Figure A.1: These are the includes required for this code to compile. Several of the functions found in the following code will not work and will throw compiler errors if these includes have not been place at the top.

```
inline double fRand(double fMin, double fMax)
{
    return fMin + ((double)rand()) / (RAND_MAX/(fMax - fMin));
}
```

Figure A.2: This is the randomization function used to initialize neural network weights in the readout layer / multinomial logistic regression.

```
class Readout{

    private:

        std::default_random_engine generator;

        /*These variables represent dimensionality of the readout*/
        int X_n;
        int W_m;
        int W_p;

        /*learning rate and regularization term*/
        double learning_rate;
        double reg;

    public:

        std::vector <double>max_vals;
        std::vector <double>min_vals;

        /*These are the learnable parameters we need to store*/
        std::vector <double> bias;
        std::vector <std::vector <double> > Wo;

        /*This is for storing parameters at lowest train loss*/
        std::vector <std::vector <double> > best_Wo;
        std::vector <double> best_bias;

        /*Used for the backwards pass...*/
        std::vector <std::vector <double> > weightGrads;
        std::vector< std::vector <double> >YMinusT;
        std::vector <double> biasGrads;

        /*input to forward pass gets copied to reservoirOuts*/
        std::vector <std::vector <double> > reservoirOuts;

        /*inputZ represents input to the softmax layer*/
        std::vector <std::vector <double> >inputZ;

        /*outputY is the output vector of the softmax layer*/
        std::vector <std::vector <double> >outputY;

        ...
```

Figure A.3: Readout class.

```
class Readout{

    public:

        ...

        /*The constructor initializes the weight matrix Wo(output)*/
        /*This classifier utilizes softmax regression.*/
        /*This should be #reservoirOutputs by #categories*/
        /*In the case of iris, this is Nx3 */
        Readout(int inputN, int outputN, double learn_rate)
            : W_m(inputN), W_p(outputN), learning_rate(learn_rate){

            X_n = 1;

            /*Regularization parameter*/
            reg = 1;

            /*Bengio recommended initial weights*/
            //TODO: make this optional
            //double r = 4*sqrt(6.0/(inputN+outputN));

            max_vals.resize(W_m, -DBL_MAX);
            min_vals.resize(W_m, DBL_MAX);

            bias.resize(W_p, 1);
            best_bias.resize(W_p, 0);

            std::vector <double> startVec(outputN, 0);
            for(int i=0; i<inputN; i++){
                Wo.push_back(startVec);
                best_Wo.push_back(startVec);
                for(int j=0; j<outputN; j++){

                    //bengio weights
                    //Wo[i][j] = fRand(-r, r);
                    Wo[i][j] = fRand(-0.1, 0.1);
                }
            }

        }

    ...
```

Figure A.4: This is the readout class constructor.

A.2 Readout Layer Functions

The following function blocks of code exist within the readout class following the previous block and after the elipses at the bottom. In order to re-create this class, simply place the functions within the readout class.

```
/*Temperature allows for exploration / exploitation tradeoff*/
void SoftmaxWithTemperature(double temperature){
    outputY.clear();
    double sumExp;

    for(int i=0; i<(int)inputZ.size(); i++){

        double max;

        sumExp = 0;
        max = *std::max_element(inputZ[i].begin(), inputZ[i].end());

        for(int j=0; j<(int)inputZ[i].size(); j++){
            /*Subtracting Max aids with numerical stability*/
            /*(exp can get large)*/
            sumExp+= exp((inputZ[i][j] - max)/temperature);
        }

        std::vector <double> tmp;
        for(int j=0; j<(int)inputZ[i].size(); j++){
            tmp.push_back(exp((inputZ[i][j]-max)/temperature)/sumExp);
        }
        outputY.push_back(tmp);

    }
}
```

Figure A.5: This is the softmax function. Simply pass a value of 1 for temperature in order to have a regular softmax function.

```
/*Multiplies the weights by the reservoir state vector*/
void forward(std::vector <std::vector <double> >&reservoirOutput){

    reservoirOuts = reservoirOutput;

    inputZ.resize(reservoirOuts.size(), std::vector<double>(W_p));
    X_n = reservoirOuts.size();

    /*for all reservoir outputs and all classes...*/
    /*Multiply the relevant reservoir output by the edge weight*/
    /*Add the result to the related output unit in inputZ*/

    for(int i=0; i<X_n; i++){
        for(int j=0; j<W_p; j++){
            double sum = 0.0;
            for(int k=0; k<W_m; k++){
                sum += reservoirOuts[i][k] * Wo[k][j];
            }

            /*This part needs to change to handle matrix output*/
            inputZ[i][j] = sum;
        }
    }

    /*Add the bias*/
    /*This also needs to change to handle matrices*/
    for(int i=0; i<(int)inputZ.size(); i++){
        for(int j=0; j<(int)inputZ[i].size(); j++){

            inputZ[i][j] += bias[j];
        }
    }

    softmax();
}
```

Figure A.6: This is the forward function. It propagates the input data vector forward by matrix multiplying with the network weights. Call this before calling the backwards function.

```
/*Loop over weight matrix, figure out derivative...*/
/*...of cross-entropy w.r.t. ij*/
/*Also update the bias*/
/*correct is the correct label for the current instance*/
void backward(std::vector<double> &labels){

    if(labels.size() != outputY.size()){
        fprintf(stderr, "ERROR (backward): labels.size()"
        "(%d) != #samples (%d)\n",
        (int)labels.size(), (int)outputY.size());

        printY();
        exit(1);
    }


    /*This computation changes for RL*/
    ComputeWeightGradient(labels);

    /*This computation changes for RL*/
    ComputeBiasGradient();

    /*This stays the same*/
    for(int i=0; i<(int)Wo.size(); i++){
        for(int j=0; j<(int)Wo[i].size(); j++){
            Wo[i][j] = Wo[i][j] - (learning_rate * weightGrads[i][j]);
        }
    }

    for(int i=0; i<(int)bias.size(); i++){
        bias[i] = bias[i] - (learning_rate * biasGrads[i]);
    }

}
```

Figure A.7: This is the backwards pass of the neural network. Must be called after the forward pass. It updates both the regular weights and the bias weights.

```
/*diagonalize and flatten*/
/*This only works for one softmax at a time*/
std::vector<std::vector<double> > diagflat(
    std::vector<std::vector<double> > &mat){

    int size = mat[0].size();
    std::vector<std::vector <double> > tmp(size,
        std::vector<double>(size, 0));

    for(int i=0; i<size; i++){
        tmp[i][i] = mat[0][i];
    }

    return tmp;
}
```

Figure A.8: This creates a diagonal matrix from the passed in vector.

```
/*multiply two matrices*/
std::vector<std::vector<double> > mat_mult(
    std::vector<std::vector<double> > &mat1,
    std::vector<std::vector<double> > &mat2){

    std::vector<std::vector<double> > ret(mat1.size(),
        std::vector<double>(mat2[0].size(), 0));

    for(int i=0; i<(int)mat1.size(); i++){
        for(int j=0; j<(int)mat2[0].size(); j++){
            for(int k=0; k<(int)mat1[i].size(); k++){
                ret[i][j] += mat1[i][k] * mat2[k][j];
            }
        }
    }

    return ret;
}
```

Figure A.9: This performs a matrix multiply.

```
/*This function takes a vector of gradients from ...*/
/*an entire episode, and associated rewards.*/
/*The size of each will be equivalent to...*/
/*the number of steps taken in the episode.*/
/*Readout weights are updated based on the REINFORCE algorithm.*/
void UpdateWeightsREINFORCE(
    std::vector< std::pair< std::vector<std::vector<double> > * ,
    std::vector< std::vector<double> > * > > &grads,
    std::vector<double> &rewards, double gamma){

    /*For all gradients...*/
    for(int i=0; i<(int)grads.size(); i++){

        /*expected future reward*/
        double efr = 0;

        /*First, we need to compute the expected future reward.*/
        for(int j=i; j<(int)rewards.size(); j++){
            efr += rewards[j] * pow(gamma, j-i);
        }

        /*(learn_rate * gradient * efr)*/
        scalar_mat_mult(learning_rate*efr, *(grads[i].first));
        scalar_mat_mult(learning_rate*efr, *(grads[i].second));

        /*Update the weights & regularize*/
        sum_mats(Wo,*(grads[i].first));

        for(int b=0; b<bias.size(); b++){
            bias[b] += (*(grads[i].second))[0][b];
        }

        //If you want to regularize, do it here.
        //std::vector<std::vector<double> > tmp2 = ...
        //scalar_mat_mult(reg, Wo);
        //Wo = subtract_mats(tmp, tmp2);

        //delete the vector of vectors... important.
        delete grads[i].first;
        delete grads[i].second;
    }
}
```

Figure A.10: This function implements the REINFORCE policy gradients algorithm.

```
/*Compute SoftmaxGradients*/
std::pair<
    std::vector<std::vector<double> > * ,
    std::vector< std::vector<double> > * >
    ComputeSoftmaxWeightGrad(int action, int debug){

    std::vector<std::vector<double> > soft_transposed;
    soft_transposed = transpose(outputY);

    std::vector<std::vector<double> > jacobian = diagflat(outputY);
    std::vector<std::vector<double> > mult_result =
        mat_mult(soft_transposed, outputY);

    //The result of this is the Jacobian
    /*diagflat relies on mat being 1xN*/
    subtract_mats(jacobian, mult_result);

    scalar_mat_div(outputY[0][action], jacobian);

    //bias grads will be 1xN vector of 1's * jacobian
    std::vector<std::vector<double> > bias_tmp(
        1, std::vector<double>(bias.size(), 1));

    std::vector<std::vector<double> >* bias_grads =
        new std::vector<std::vector<double> >(1, jacobian[action]);


    std::vector<std::vector<double> > tmp;
    tmp.push_back(jacobian[action]);

    std::vector<std::vector<double> > tposed_resOuts =
        transpose(reservoirOuts);

    std::vector<std::vector<double> > * gradients =
        new std::vector<std::vector<double> >(
            mat_mult(tposed_resOuts, tmp));

    return std::make_pair(gradients, bias_grads);
}
```

Figure A.11: This function implements the gradient computation based on the softmax function. This is for use with policy gradients.

```
/*Compute the Weight Matrix Gradient*/
/*Currently, this assumes a cross-entropy loss function.*/
/*Computing derivative of cross-entropy loss w.r.t. weights.*/
void ComputeWeightGradient(std::vector<double>&labels){

    weightGrads.resize(W_m, std::vector<double>(W_p));
    YMinusT.clear();

    /*This could be labels.size(), but they should be the same.*/
    for(int i=0; i<(int)outputY.size(); i++){

        std::vector <double> tmp;
        for(int j=0; j<(int)outputY[i].size(); j++){
            if(j == labels[i])
                tmp.push_back(outputY[i][j] - 1);
            else
                tmp.push_back(outputY[i][j]);
        }
        YMinusT.push_back(tmp);
    }

    /*Multiply reservoirOuts transposed by the YMinusT vector*/
    reservoirOuts = transpose(reservoirOuts);

    for(int i=0; i<X_n; i++){
        for(int j=0; j<W_p; j++){
            for(int k=0; k<W_m; k++){
                weightGrads[k][j] += reservoirOuts[k][i]
                                        * YMinusT[i][j];
            }
        }
    }

    for(int i=0; i<(int)weightGrads.size(); i++){
        for(int j=0; j<(int)weightGrads[i].size(); j++){
            weightGrads[i][j] /= (double)X_n;
        }
    }

    /*uncomment for experimental regularization*/
    //weightGrads = sum_mats(weightGrads,
                        (scalar_mat_mult(reg, Wo)));
}
```

Figure A.12: This function implements the computation of gradients based on the derivative of cross-entropy loss. This is used in classification problems.

```
/*Nudge the weights by epsilon*/
/*Compute numerical derivative (i.e. limit form)*/
/*(E(Wo + epsilon) - E(Wo - epsilon)) / 2epsilon */
double CheckGrads( std::vector< std::vector<double> > &inputs,
    std::vector<double>&labels, double epsilon){

    /*This needs to be a loop within which we nudge a weight,*/
    /*forward prop, compute numerical, reset, do next weight*/
    std::vector <double> numerical_gradients;
    save_Wo();
    for(int i=0; i<(int)Wo.size(); i++){
        for(int j=0; j<(int)Wo[i].size(); j++){
            /*positive nudge, negative nudge*/
            set_to_saved();
            Wo[i][j] += epsilon;
            forward(inputs);
            double Jpositive = CrossEntropy(labels);
            set_to_saved();
            Wo[i][j] -= epsilon;
            forward(inputs);
            double Jnegative = CrossEntropy(labels);
            double num_grad = (Jpositive - Jnegative)
                                / (2.0 * epsilon);
            numerical_gradients.push_back(num_grad);
        }
    }

    ...
}
```

Figure A.13: This function implements part 1 of gradient checking. This ensures the gradient checking computes the numerical limit formulation of a gradient in order to compare against the derived version.

```
/*Nudge the weights by epsilon*/
/*Compute numerical derivative (i.e. limit form)*/
/*(E(Wo + epsilon) - E(Wo - epsilon)) / 2epsilon */
double CheckGrads( std::vector< std::vector<double> > &inputs,
    std::vector<double>&labels, double epsilon){
    ...

    /*Need to compute the norm...*/
    /*reset weights*/
    set_to_saved();
    std::vector <double> grad_concat = mat_to_vec(weightGrads);
    double numerator = 0, denominator = 0, norm = 0;
    for(int i=0; i<(int)numerical_gradients.size(); i++){
        numerator += pow(grad_concat[i] - numerical_gradients[i], 2);
    }
    numerator = sqrt(numerator);

    double tmp = 0;
    for(int i=0; i<(int)numerical_gradients.size(); i++){
        tmp += pow(grad_concat[i], 2);
    }
    denominator += sqrt(tmp);

    tmp = 0;
    for(int i=0; i<(int)numerical_gradients.size(); i++){
        tmp += pow(numerical_gradients[i], 2);
    }

    denominator += sqrt(tmp);
    norm = numerator / denominator;

    if(norm < epsilon){
        printf("norm (PASSED): %f epsilon: %f\n", norm, epsilon);
    }else{
        printf("norm (FAILED): %f epsilon: %f\n", norm, epsilon);
    }

    return 0;
}
```

Figure A.14: This function implements part 2 of gradient checking. This ensures the gradient checking computes the numerical limit formulation of a gradient in order to compare against the derived version.

Vita

John Jeffrey McClain Reynolds was born on July 26th, 1992 in Nashville, TN. After graduating from high school in 2011, he attended the University of Tennessee at Knoville. Majoring in computer science, he graduated in 2015 with a bachelor's degree. He completed his master's degree in 2018, and subsequently his doctorate with a focus on machine learning (neural networks) in 2019.