Neuromorphic Computing with Reservoir Neural Networks on Memristive Hardware

COSC460 Research Project

Aaron Stockdill
68299033
aas75@uclive.ac.nz

Under the supervision of
Dr Kourosh Neshatian
kourosh.neshatian@canterbury.ac.nz

Department of Computer Science and Software Engineering
University of Canterbury
Christchurch, New Zealand
14 October 2016

Abstract

Building an artificial brain is a goal as old as computer science. Neuromorphic computing takes this in new directions by attempting to physically simulate the human brain. In 2008 this goal received renewed interest due to the memristor, a resistor that has state, and again in 2012 with the atomic switch, a related circuit component. This report details the construction of a simulator for large networks of these devices, including the underlying assumptions and how we model specific physical characteristics. Existing simulations of neuromorphic hardware range from detailed particle-level simulations through to high-level graph-theoretic representations. We develop a simulator that sits in the middle, successfully removing expensive and unnecessary operations from particle simulators while remaining more device-accurate than a wholly abstract representation. We achieve this with a statistical approach, describing distributions from which we draw the ideal values based on a small set of parameters. This report also explores the applications of these memristive networks in machine learning using reservoir neural networks, and their performance in comparison to existing techniques such as echo state networks (ESNs). Neither the memristor nor atomic switch networks are capable of learning time-series sequences, and the underlying cause is found to be restrictions imposed by physical laws upon circuits. We present a series of restrictions upon an ESN, systematically removing loops, cycles, discrete time, and combinations of these three factors. From this we conclude that removing loops and cycles breaks the "infinite memory" of an ESN, and removing all three renders the reservoir totally incapable of learning.

Contents

List of Symbols and Notation

General Mathematical Notation

$\circ$	Function composition, e.g. $f(g(x)) = (f \circ g)(x)$.
$\lightning$	Contradiction.
Fv	Filter (or operator) application. Filter F maps function v between vector spaces.

Machine Learning Constants and Operations

$\mathrm{act}(c)$	The set of instances of a training set that truly belong to class c.
$\mathbf{u}(t)$	The input vector at time t.
$\mathcal{N}(\cdot)$	The neighbours of a vertex in a graph (or graph-like structure).
$\mathbf{y}(t)$	The output vector at time t.
$\mathrm{pred}(c)$	The set of instances of a training set that are predicted to belong to class c.
τ	The number of training instances, or the Mackey-Glass "difficulty" parameter.
$\mathbf{W}$	Matrix of weights connecting neurons, where w_{ij} connects neuron j to i.
$\mathbf{x}(t)$	A vector representing the reservoir state at time t.

Neuromorphic Constants and Operations

$\alpha, \beta, \gamma \ldots$	Greek letters represent physical constants/properties.
G	Conductance, the inverse of resistance.
$\mathcal{I}$	The set of input groups.
ℓ	The length across a gap between groups, measured in particle diameters.
$\mathcal{O}$	The set of output groups.
p	The coverage proportion of the board.
p_c	Percolation threshold, the coverage proportion of the board before "short-circuiting."
V_T, I_T	Threshold voltage and current, respectively.

Vectors and Vector Operations

$\mathbf{A}, \mathbf{B}, \mathbf{C} \ldots$	Uppercase bold-face latin letters represent matrices.
$\mathbf{a}, \mathbf{b}, \mathbf{c} \ldots$	Lowercase bold-face latin letters represent vectors.
$\lvert\cdot\rvert$	The dimension(s) of a vector or matrix. For example, if $\mathbf{x} \in \mathbb{R}^3$, then $\lvert\mathbf{x}\rvert = 3$. Also absolute value of scalars, and the cardinality of a set.
$\tanh(\cdot)$	Hyperbolic tangent, applied element-wise.
$[\cdot;\cdot]$	Vertical concatenation of vectors.
$\lVert\cdot\rVert_2$	The Euclidean norm of a vector. What is commonly considered the length.

1 Introduction

"The scientist is not a person who gives the right answers, he's one who asks the right questions."
— Claude Lévi-Strauss

Modern computing is increasingly turning to artificial intelligence and machine learning to accomplish the evermore ambitious goals set before it. To build a machine that can attain the "gold-standard" of learning—that is, to match a human brain—is the ultimate goal of researchers around the world. A human brain is able to perform better than modern computers at deceptively "simple" tasks such as object recognition, while using in the order of a *millionth* of the power: hence the allure of machines that can match it. Recent advances in neuromorphic computing have meant renewed interest in the hopes of building such a machine [27].

Neuromorphic computing is a cross-disciplinary field incorporating researchers from Computer Science, Physics, Mathematics, and Statistics, all working together to build what they hope will be a machine capable of matching a human brain. In this report we focus our attention on the recent development of memristive hardware, using novel fundamental circuit components to construct "intelligent hardware".

Memristors, and their close cousins atomic switches, are both varieties of memristive hardware—that is, hardware which changes itself based on its own past. This ability to remember could help unlock new advances in machine learning, by moving the learning into the hardware. Because they are so new, a significant amount of research is needed before the utility of memristive hardware becomes clear.

In this research, we explore the learning potential of proposed homogeneous memristive hardware. We build upon a model of learning called reservoir neural networks, and explore the kinds of problems that memristive hardware acting as a reservoir will be able to solve. As will become clear, we also discuss the physical limitations of such hardware, and what it means to attempt to apply machine learning techniques to physical hardware. Some consideration of how to overcome these issues, as well as the next steps in research, are also presented.

1.1 Motivation

The Nanotechnology Research Group at the University of Canterbury Physics and Astronomy Department have been working on constructing a network of atomic switches, a type of memristive circuit component. Their work is reaching the point that hardware has now been produced, and initial understanding of the dynamics of these components is available.

Much has been written about the potential of memristive hardware in machine learning contexts, but the work to date has been done in a large part by physicists. This leaves open a chance for an in depth discussion of how machine learning can be mixed with novel hardware environments, and how the features of the hardware impact on learning. In particular, the exact features of this hardware are still unknown. Because the hardware is so different to the computing hardware we are familiar with, the types of behaviour we might see are uncertain.

A large portion of the existing literature focuses on the memristor, but less work has been done on the atomic switch. Because this is the hardware that the University of Canterbury is invested in, understanding how the memristor and atomic switch differ can direct future work, and inform how existing literature can be understood in terms of atomic switches.

1.2 Goals

The goals of this research project are threefold: first, simulate large networks of memristive devices efficiently and rapidly; second, incorporate these networks of memristive devices into reservoir learning methods and demonstrate their learning potential; and third, compare how memristors and atomic switches relate to each other and traditional reservoir neural networks in respect to reservoir learning. A subsequent goal emerged during the project, which was to determine which features of reservoir neural networks were most important in learning, particularly in learning temporal data sets.

When this work started, there was an understanding that physical hardware would be available for experimentation. This would mean we would be able to provide the first in depth understanding of memristive hardware outside of simulations. However, early discussions revealed that the atomic switches as they currently stand are not suitable for use in machine learning tasks. The combination of long write times and inconvenient environment make them slow and inaccessible, and so we abandoned the goal of implementing reservoir learning on physical hardware for practical reasons.

1.3 Organisation

This report opens with a brief summary of the background material required for the content discussed in later chapters. We also provide a brief overview of existing work in the field of neuromorphic computing with memristive hardware, and the work that inspired this project, covered in Chapter 2. The novel contributions from this project are presented in Chapters 3, 4, and 5. Chapter 3 covers simulating the memristive and atomic switch hardware, while Chapter 4 focusses on applying reservoir neural network machine learning techniques to the simulations. Chapter 5 presents the capabilities of homogeneous memristive hardware in a reservoir paradigm. The underlying causes are discussed, and the implications of these results explored. We conclude this report with Chapter 6, in which we present a summary of the research, the limitations of the work, and the future research avenues and questions raised by this project.

Background & Literature Review 2

"If I have seen further it is by standing on the shoulders of giants."
— Sir Isaac Newton

This interdisciplinary project draws on machine learning, mathematics, statistics, and physics. Because of the broad scope and significant background knowledge required, this section provides a brief overview of each of the key concepts, followed by a summary of the directly related literature. The background work occurred over the course of decades, with each field working in parallel, often independently and sometimes influencing each other. The presentation of information here is in order of concept progression, but the interconnected nature means a full understanding may require repeated readings.

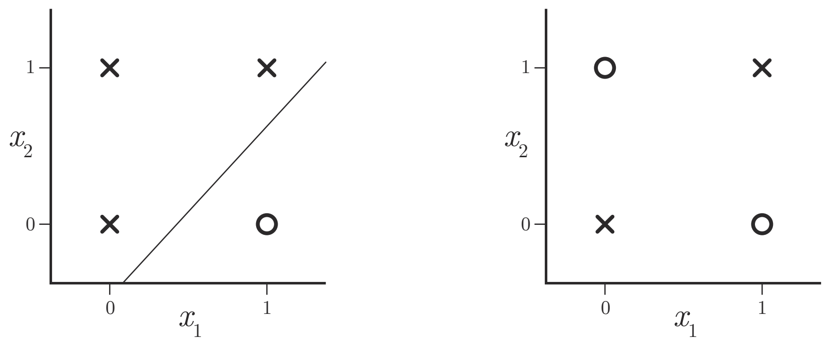
2.1 Machine learning

Since the invention of computers, there has been a desire to make them think like a person. In 1956, John McCarthy introduced the term *artificial intelligence* to the world, and proposed a two-month workshop to build an intelligent machine [32, Section 1.3.2]. Sixty years later, we are not a lot closer to that original goal—but in those sixty years, we have achieved the incredible: self-driving cars, grandmaster Chess and Go players, and countless other significant achievements.

Machine learning is an area that has significant overlap with artificial intelligence. Responsible for a significant portion of the results listed above, machine learning has its roots in mathematics and statistics [3]. This rigorous approach to intelligence has led to a change in expectations—the desire to build a thinking, "aware" intelligence has diminished (although certainly has not disappeared [12]), and a new goal has come to the fore, the goal of building a machine capable of identifying patterns and learning from datasets.

One prominent machine learning technique of the past decade is the *neural network*. The neural network is conceptually simple: the best model for intelligence we have is a human brain; the human brain is a network of neurons; ergo, to build an intelligent machine we should build a network of neurons. The simplest network consists of a single neuron, providing a starting point for significant future research.

The *perceptron* was an early model of learning, modelling a single neuron that took some inputs, and produced an output signal in response [3, Section 4.1.7]. Equation (2.1) summarises their function, wherein $\phi(\cdot)$ is a fixed transformation from an input $\mathbf{x}$ into a feature vector $\phi(\mathbf{x})$, $f(\cdot)$ is the activation function frequently defined as in Equation (2.2), and $\mathbf{w}$ is a vector of weights that is updated according to some function, often some variety



(a) The "and" problem, $x_1 \wedge x_2$, with one possible solution illustrated.

(b) The "xor" problem, $x_1 \oplus x_2$, which is *not* linearly separable.

Figure 2.1: Two decision problems with the same concept: given inputs x_1 and x_2, classify it as either `true` (noughts) or `false` (crosses). No straight line will split the noughts and crosses in (b).

of gradient descent.

$$y(\mathbf{x}) = f(\mathbf{w}^\top \phi(\mathbf{x})) \tag{2.1}$$

$$f(a) = \begin{cases} +1 & \text{if } a \geq 0 \\ -1 & \text{otherwise} \end{cases} \tag{2.2}$$

These single perceptrons were effective learners, and were simple enough to train using gradient descent. Perceptrons had important limitations—they were found to be equivalent to a linear classifier, and thus limited to finding a linear decision boundary. Figure 2.1 shows an example of linear and nonlinear decision boundaries.

To overcome this linear boundary limitation of perceptrons, the first neural networks were developed—the multilayer perceptron. This involved stacking sets of perceptrons upon one another, feeding the outputs of the previous layer as inputs into the next. The activation function f is abandoned for a new continuous (and hence differentiable) function, often either tanh or the *sigmoid* function,

$$\sigma(a) = \frac{1}{1 + e^{-a}}. \tag{2.3}$$

In such a way, the first neural networks were constructed, and are today referred to as *feed-forward neural networks.* Feed-forward neural networks are capable of learning nonlinear decision boundaries, and are relatively easy to train. The back-propagation algorithm, based around Equation (2.4), is able to update the internal weights of the network similarly to how gradient descent will update the weights of a single perceptron [32, Section 18.7.4].

$$w_{ij} \leftarrow w_{ij} + \alpha \times a_i \times \Delta_j \tag{2.4}$$

$$\Delta_i = \sigma'(in_i) \times \begin{cases} (y_i - a_i) & \text{if } i \text{ is an output neuron} \\ \sum_j w_{ij}\Delta_j & \text{otherwise} \end{cases} \tag{2.5}$$

$$a_j = \begin{cases} x_j & \text{if } j \text{ is an input neuron} \\ \sigma(\sum_i w_{ij} a_i) & \text{otherwise} \end{cases} \tag{2.6}$$

The variable w_{ij} represents the weight of the connection between neuron i and neuron j, a_i represents the output of neuron i, α is the learning rate of the network, and x_i and y_i are elements of the input and output vectors, respectively.

Despite resolving the issue of the nonlinear decision boundary, feed-forward neural networks were not a panacea. As the neural network showed its power, with more and more expected of it, we encountered a significant downside. Feed-forward neural networks are able to learn mathematically pure functions, but cannot learn temporal functions—that is, functions that have hidden dependencies on time and state. Although this can be worked around by encoding the time or state into the input, it becomes difficult as the domain becomes complex.

The solution is to remove the feed-forward restriction, and hence allow cycles to occur in the network. This enables information to loop within the network, meaning that there is now an implicitly encoded *state*. The network is now able to produce output based on both the current input and on a knowledge of past inputs. The cost of this power is training—back-propagation by itself is no longer a viable training method, because the local minima for a certain input move over time.

There are three main recurrent neural network training algorithms, the most common being *back-propagation through time*. This method involves "unfolding" the network through time by taking the output of the network at time t, and combining it with the input for time $t + 1$. This creates large networks that are difficult to train, and is much more susceptible to local minima traps [24]. A similar method is real-time recurrent learning, which functions much like traditional back-propagation, but estimates the gradient because of the difficulty of calculating it [6]. Perhaps the most successfully applied training method is the extended Kalman filter. The extended Kalman filter performs linearisation around the working point, and then applies a regular Kalman filter. A regular Kalman filter works by mapping not points, but entire distributions, and so is more tolerant to variation in the data [6]. Lastly there is the reservoir neural network.

2.2 Reservoir neural networks

A reservoir neural network builds on the concept of a recurrent neural network, similarly allowing cycles in the neurons, encoding state in the network itself. One significant difference is that the weights of the connections are no longer updated. The weights are static, and instead the training occurs in a *readout* layer. By moving the training out of the reservoir, the highly interconnected structure can be as complex as desired, without making the training more difficult. This has the added benefit of making a significantly larger neural network, now called a *reservoir*, computationally viable to work with.

The first kind of reservoir neural network developed by Jaeger in 2001 is the *echo state network (ESN)* [16]. The learner is composed of three pieces, each represented by a matrix: an input layer $\mathbf{W}^{\text{in}}$ mapping from the input to the neurons in the reservoir, with a bias; the reservoir $\mathbf{W}$ consisting of an arbitrarily connected set of neurons, which are allowed to form cycles and loop back on themselves; and a readout layer $\mathbf{W}^{\text{out}}$ which takes the input and the state of all the neurons, and learns a mapping to the expected output [16]. This structure is visible in Figure 2.2. The full formal definition is

$$\begin{aligned}
\mathbf{y}(t) &= \mathbf{W}^{\text{out}}[1; \mathbf{u}(t); \mathbf{x}(t)] \\
\mathbf{x}(t) &= (1-\alpha)\mathbf{x}(t-1) + \alpha\tilde{\mathbf{x}}(t) \\
\tilde{\mathbf{x}}(t) &= \tanh\left(\mathbf{W}^{\text{in}}[1; \mathbf{u}(t)] + \mathbf{W}\mathbf{x}(t-1)\right).
\end{aligned} \tag{2.7}$$

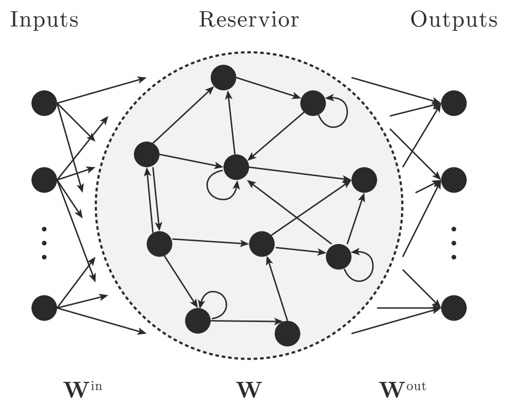


Figure 2.2: A diagram of how an ESN is laid out, with the input, reservoir, and readout layers. The matrix labels are in reference to the arrows.

The operator $[\cdot;\cdot]$ is a vertical concatenation of vectors. The vector $\mathbf{u}(t)$ is the input and $\mathbf{x}(t)$ is the internal state of the reservoir at time t. $\mathbf{y}(t)$ is the output from the reservoir at time t. The parameter α is the *leaking rate*, determining the mix of old and new information in the network. The hyperbolic tangent $\tanh(\cdot)$ is applied element-wise.

A defining feature of the ESN is the *echo property* [16]. This property guarantees that the current output is dependent on the history of inputs, and for a long enough history of inputs, the current state is unique. That is, the current state is a pure injective function E on all previous inputs:

$$\mathbf{x}(t) = E(\dots, \mathbf{u}(t-1), \mathbf{u}(t)). \tag{2.8}$$

This is because the network acts as a kind of fading memory, basing the output on all information but giving most weight to recent inputs. There is no easy way to determine whether an arbitrary network satisfies the echo property: certain known conditions are *sufficient* but not *necessary*—notably that the *spectral radius*, the maximum absolute eigenvalue, is less than one.

The readout layer does all the learning, trained through any least-squares matrix solution. In this project, we will use ridge regression, also called Tikhonov regularisation [24], defined as

$$\mathbf{W}^{\text{out}} = \mathbf{Y}^{\text{target}}\mathbf{X}^{\top}\left(\mathbf{X}\mathbf{X}^{\top} + \beta\mathbf{I}\right)^{-1}. \tag{2.9}$$

The regularisation constant β is used to penalise large $\mathbf{W}^{\text{out}}$. There are an infinitude of different parameters to tune for individual problems, but in general the key parameters are the leaking rate, the spectral radius of $\mathbf{W}$, and input scaling from $\mathbf{W}^{\text{in}}$.

Although they have a seemingly complicated definition, ESNs are simpler to program and faster to train than traditional recurrent neural networks. Surprisingly, and pleasingly, ESNs are no less powerful than the other recurrent neural network training techniques [7]. Thus the improvements in training time come at no cost of learning potential, and so make a solid starting point for this research project.

ESNs are powerful learning systems, but remove most restrictions upon the reservoir design. This means they can be arbitrarily complex, and while their simple training usually does not make this a problem, they can suffer from having a lot of hyperparameters to tune. To combat this, Čerňanský and Tiňo proposed a restricted form of ESN called the *feed-forward ESN* [6]. The feed-forward ESN resembles a regular ESN, except a restriction is placed on connections in the reservoir. For a reservoir with n neurons, there must be a single chain of length n containing every neuron, there must be no cycles, and for some numbering of neurons $1 \dots n$ neuron i may connect only to neurons $j > i$.

A feed-forward ESN is not, in the traditional sense, feed-forward. This is because at time t a neuron is still aware of the input at time $t-1$ by connections from previous neurons. Thus the network still has a state, which a true feed-forward neural network does not. But the memory encoded in the network no longer extends back to the beginning of input, and is now limited to at most n steps back in time [6]. Because of this, the network is now equivalent to a feed-forward neural network with explicit memory input for the past n steps.

Equivalent to the ESN is the liquid state machine (LSM). The LSM was originally defined by Maass et al. in 2002 in terms of an input filter—a liquid which holds the state driven by the previous inputs—and an output layer [25]. Although not necessarily implemented using neural networks, Maass et al. provided a realisation of the liquid using integrate-and-fire (or *spiking*) neurons.

The LSM is attractive because it has "universal computational power for time-varying inputs" [25]. However, to achieve this result, LSMs place some strict demands on the input filters (a separation property, see definition 2.1) and readout layer (an approximation property, see definition 2.2), making their real-world generality harder to guarantee. For this reason, the LSM was not chosen as the basis of the learning algorithm for this project. Work that does so would be a useful extension on this project, to see if working from the reference point of the LSM model produces different results.

Definition 2.1 (Separation property [25])
A class CB of filters has the point-wise separation property with regards to input functions from U^n if, for any two functions $u(\cdot), v(\cdot) \in U^n$, such that for some $s \leq 0$ we have $u(s) \neq v(s)$, then there exists some $B \in CB$ that separates $u(\cdot)$ and $v(\cdot)$, that is $(Bu)(0) \neq (Bv)(0)$.

Definition 2.2 (Approximation property [25])
A class CF of functions has the approximation property if, for any $m \in \mathbb{N}$, any closed and bounded (i.e. compact) set $X \subseteq \mathbb{R}^m$, any continuous function $h : X \to \mathbb{R}$, and any given $\rho > 0$ then there exists some $f \in CF$ such that $|h(x) - f(x)| \leq \rho$ for all $x \in X$. Multidimensional outputs are defined similarly.

2.3 Neuromorphic computing

In 1989, Mead coined the term *neuromorphic computing* to mean software and hardware that behave like a biological neural network—like a brain [26]. The reason for this is clear: the human brain is the gold standard of intelligence. There is no other system like it that is as capable of massive, parallel computation of tasks that currently seem computationally impossible. And the human brain does all this with just tens of watts of power. A comparable computer today needs tens of *giga*watts [35]. At a glance this would make neuromorphic computing a subfield of artificial intelligence, but the broad scope of these brain-making projects ensure it is a field of its own, drawing researchers from computer science and engineering, mathematics and statistics, and psychology and neurology.

In present computers, we use what is known as the von Newmann architecture. This model of computers separates the processing hardware (i.e. the CPU) from the memory hardware (i.e. RAM and disk), in much the same way that a Turing machine separates the logic inside the state machine from the input, output, and memory contained in the tape. Neuromorphic computing breaks down this distinction, and instead blends together the two concepts, maintaining state (or memory) within the processing units [13]. This removes the

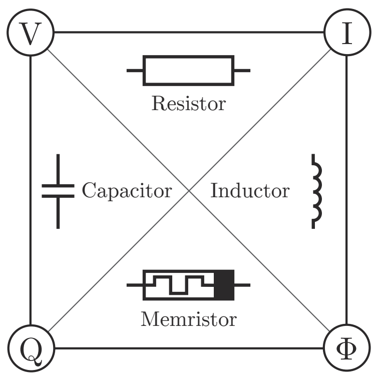


Figure 2.3: The fundamental circuit components linking the four properties of an electrical circuit. The memristor provides the link between charge and flux.

bottleneck seen today in computer systems, when data must be shunted to and from memory, or even worse from disk.

In an attempt to reach the power efficiency of the human brain, hardware implementations of neural networks became popular. The approaches are varied, ranging from the spiking neural network architecture (SpiNNaker) project from the University of Manchester [10], which uses thousands of processing cores, each of which models thousands of neurons, through to the TrueNorth chip from IBM [1], which abandons what is currently considered a "computer chip" by combining millions of transistors into *neurosynaptic cores*. Both of these projects arrange existing hardware in novel ways in an attempt to create the massive interconnected network found in a brain.

The difficulty with this approach is that existing hardware does not closely resemble cells we find in a brain. Brain cells are self-updating, contain their own state, and generally use energy only when firing. As such, attention is turning away from current hardware components, and instead towards new types of hardware that resemble brain cells. This hardware must be small enough to pack densely, cheap enough to make millions, and consist of these updating, power-efficient characteristics sought-after by neuromorphic engineers. Some research has been directed into custom silicon solutions [14], but more interesting is the goal to use fundamental circuit components. Two candidates have appeared: the memristor, and the atomic switch.

2.4 Memristors and atomic switches

Electrical circuits traditionally consist of three fundamental components: resistors, capacitors and inductors. These components create relationships between current, voltage, charge, and flux. Charge is the integral of current through time, and flux is the integral of voltage through time. Thus four relationships are possible, avoiding relating current with charge and voltage with flux. The final, previously missing link between charge and flux is filled by a component known as the *memristor*, first theorised in 1971 by Chua [8]. Figure 2.3 illustrates these relationships. In May 2008, Strukov et al. discovered the first memristors forming naturally at nanometre scales [41].

The family of memristors is defined by a relationship between voltage and current with a function $R(\cdot,\cdot)$, shown in Equation (2.10), which is in turn specified by a differential equation,

Equation (2.11) [41].

$$V = R(x, I) \cdot I \tag{2.10}$$

$$\frac{dx}{dt} = f(x, I) \tag{2.11}$$

The function $f(\cdot, \cdot)$ is device-specific, and left undefined. The variable x is a *state variable*, used as a mathematical analogy of the physical changes within the device. This set of equations defines a *charge-controlled* memristor, but an alternative definition is called a *flux-controlled* memristor, which is specified by analogous Equations (2.12) and (2.13).

$$I = G(x, V) \cdot V \tag{2.12}$$

$$\frac{dx}{dt} = f(x, V) \tag{2.13}$$

As before, x and $f(\cdot, \cdot)$ are left unspecified, and are device specific. Because conductance $G = 1/R$ is a more convenient definition in the context of this project, we will be working with the flux-controlled memristor definition.

The given definitions are suitable for a class of devices, allowing for different possible realisations of a memristor. For the purposes of this project, any reference to a memristor should be considered as a reference to the standard memristor [19]. The function definition of a standard memristor is

$$\frac{dx}{dt} = \beta V + \frac{1}{2}(\alpha - \beta)(|V + V_T| - |V - V_T|). \tag{2.14}$$

It makes the simplifying assumption that the state variable x is the present conductance G. The constants α, β, and V_T are specific to a device, with V_T as the threshold voltage. This threshold voltage is the point at which a memristor changes from low conductance to high conductance. When simulating a specific device, these constants are determined empirically. The standard memristor is not the only possible model, with improvements given by Querlioz et al. towards modelling specific devices [30]. Because this project does not target memristors directly, We use the standard memristor, it being a simpler and more general model.

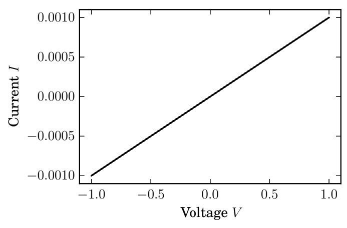
In contrast to the memristors defined above, there exists a model called an *atomic switch* [34], which will also react to voltage and current, but does so differently, switching between a high- and low-conductance state with negligible intermediate transition. The definition of atomic switches is

$$G(t+1) = \begin{cases} G_{\max} & \text{if } V(t) \geq V_T \text{ and } I(t) < I_T \\ G_{\min} & \text{otherwise.} \end{cases} \tag{2.15}$$

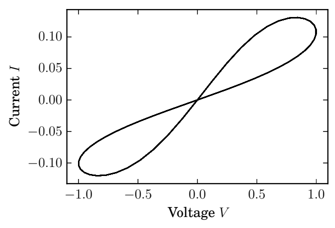
As before, V_T is a voltage threshold, and now there exists a current threshold current I_T. Such a current threshold is not strictly necessary, but was shown by Fostner and Brown to add variability, which may be useful for learning [9]. It represents the current at which a switch breaks. The constant $G_{\max}$ is the "on" state of the switch, and similarly $G_{\min}$ is the "off" state of the switch. The on state is typically set as a constant value, while the off state conductance for switch i is based on the size of the switch ℓ_i using the relation in Equation (2.16) [9].

$$G_{\min}(\ell_i) = \alpha e^{-\beta \ell_i} \tag{2.16}$$

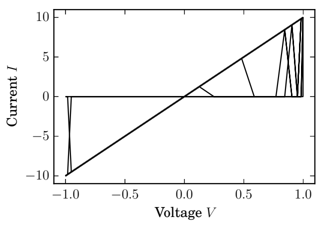
The constants α and β are device-specific, similarly to their counterparts in the memristor definition, Equation (2.14). Atomic switch networks have the advantage of being much easier



(a) Resistor



(b) Memristor



(c) Atomic Switch

Figure 2.4: The current-voltage curves of three circuit components.

to produce than memristor networks. This makes them attractive for possible applications, as they can make potentially large networks quickly and cheaply.

The behaviour of circuit components can be compared using a diagram called a current-voltage plot (abbreviated I-V). Figure 2.4 shows this plot for a resistor, a memristor and an atomic switch. A resistor, Figure 2.4a, presents a line segment, illustrating the linear relationship. The characteristic curve of a memristor is the *pinched hysteresis*, Figure 2.4b, a curve which displays a memory of past inputs by the change in gradient. A switch creates an I-V curve that resembles an angular version of the pinched hysteresis, Figure 2.4c.

2.5 Related work

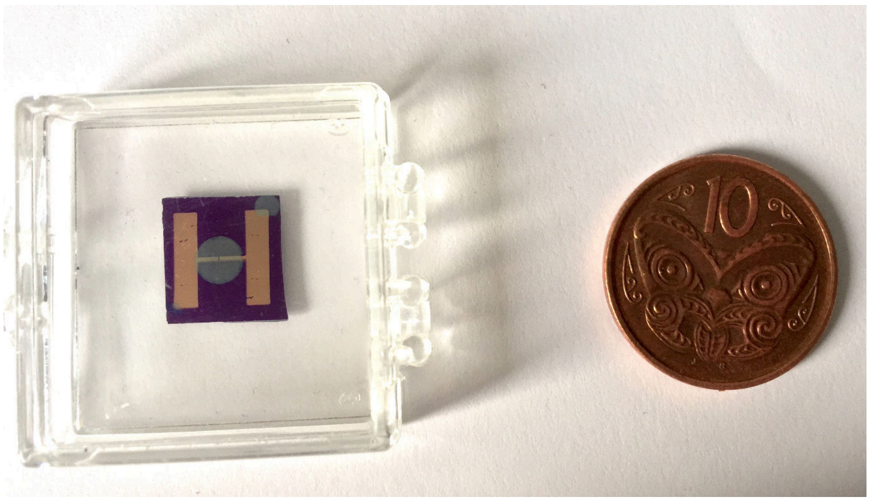
The motivation for this research project grew out of the work by the Nanotechnology Research Group at the University of Canterbury (NRG). The NRG have conducted initial tests using a percolating switch network [34], and the initial results show potential as a basis for neuromorphic hardware. They continued this work, developing and better understanding the similarities between memristors and atomic switches [9].

Atomic switches have potential similarities to memristors, most notably their conductance is a function of past inputs. Because of this they may well be suited to the same roles as memristors, including uses in neuromorphic hardware. Atomic switches are manufactured in a random way, meaning larger numbers are more easily produced. Because the resulting networks are relatively simple to make but complex in structure, there is a lot of scope for research into their behaviour.

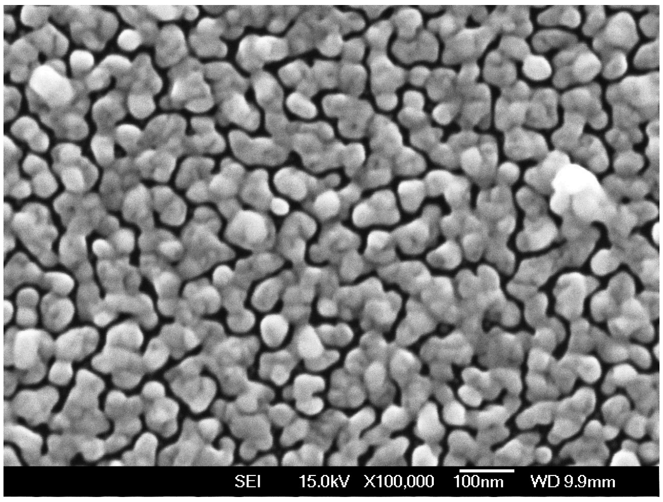
The NRG have been recently been able to manufacture some of these atomic switch networks using large magnetised vacuum chambers at temperatures below 200 K. The resulting hardware is shown in Figure 2.5a, where the gold sections are electrical contacts, and the grey is the tin particle depositions. Figure 2.5b is a picture from a scanning electron microscope showing the structure of the chip in nanometre scales. Finally we can see the setup used in construction of the chip in Figure 2.5c.

The hardware that currently exists suffers from important limitations. First, the speed at which we can read and write to the chip is severely limited to approximately one voltage change per second. Second, the number of inputs and outputs is currently limited to a single input and a single output. Third, the only information we can read from the network is the amount of current flowing through the circuit, which changes in response to the network conductance overall. Although initially this project had hoped to use this hardware, the limitations of this early-stage hardware were too significant to overcome.

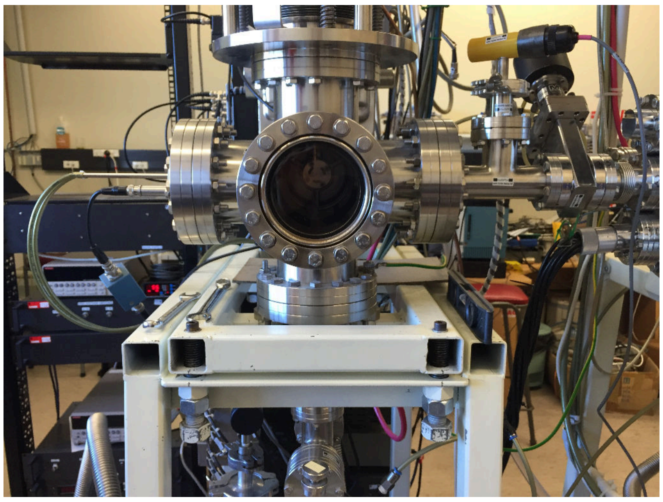
The NRG have created a series of Matlab programs to simulate the hardware, using the approach from a Masters Dissertation by Smith [38, Chapter 3], who studied in the NRG. Section 3 covers the algorithms this project uses in more detail, but the Matlab code itself



(a) The atomic switch network.



(b) A scanning electron microscope picture of the atomic switches.



(c) The setup to construct the hardware.

Figure 2.5: Images of the atomic switch network hardware.

was quickly abandoned as a viable starting point. Although concepts were borrowed, the code itself would require significant rewriting, essentially from scratch, to be appropriate for this project. In addition, the code from the NRG focuses in areas that are not of direct interest to this project—we abstract these to a higher level without loss of applicability.

Outside the University of Canterbury, other research labs have been working on competing atomic switch network architectures. One approach is to use silver nanowires, as was done by Stieg et al. [40]. These silver nanowires form and break in much the same way as the percolation networks from Sattar et al., and exhibit memristive properties suggesting potential neuromorphic applications.

Continued by Avizienis et al., silver nanowires have strongly memristive characteristics [2], including the important hysteresis I-V curve. The networks of silver nanowires also contain patterns within the network, with different sections exhibiting different patterns. This led to the hope that these could be combined using a readout layer.

Further work by Sillin et al., explored the use of reservoir computing using these silver nanowire atomic switches [37]. Their networks could be used to generate higher harmonics of the input waves, as well as generate square and triangular waves of the same frequency when sensor readings from across the network are combined. These results show how the network dynamics may be able to be used to generate new outputs from an input, a fundamental feature of reservoir learning.

While the University of Canterbury Nanotechnology Research Group has focused primarily on atomic switch networks, the majority of the literature is devoted to networks of memristors. Their theoretical properties make them the ideal self-updating neuromorphic learning component, and so work with small networks has already begun.

In 2010, Jo et al. proposed using memristors as "synapses" in neuromorphic hardware [17]. They showed that the synapse could be updated by applying voltage pulses in a specific way. This laid the groundwork for Linares-Barranco et al. and Saïghi et al. to explore spiking-time-dependent-plasticity with memristive synapses [23, 33]. Linares-Barranco et al. have also successfully constructed a self-learning visual cortex. There have been other successes, including those of Hu et al. with distorted letter recognition [11].

In parallel, Zhao et al. explored how to structure memristor networks [42]. In particular, they demonstrate that the 2-terminal devices so often considered are susceptible to alteration after training. This is because memristors do not have a "learning" mode and a "predicting" mode, but instead are always updating their conductance. To combat this, Zhao et al. design a 3-terminal memristor that is able to toggle between two modes, and so protect the memristor from learning when it should not. Such an approach is unfortunately not applicable to atomic switch networks, but is certainly interesting for memristor networks.

An important discovery was associative memory, first demonstrated by Pershin and Di Ventra in 2010 [28]. By exploiting spiking, the continually updating weights of memristors was considered a benefit, not a problem, meaning that the Hebbian philosophy of "fire together, wire together" was being realised in hardware. The canonical example shown by Pershin and Di Ventra was the Pavlov's Dog experiment. Two signals, a bell and the smell of food, are initially distinct to the learner. The smell of food is associated with a positive reward (i.e., food), and the learner is exposed to both signals simultaneously. The learner successfully associated a bell with food rewards.

Work by Indiveri et al. in 2013 provides a good summary of learning options using memristive hardware, and the challenges it currently faces [15]. They discuss learning options including probabilistic inference using Markov-Chain Monte Carlo sampling, and reservoir computing in either the ESN or LSM paradigm. Importantly, the discussion does not involve using a structured reservoir, and instead considers arbitrary networks, not necessarily those in the shape of a tidy grid, as was being used by others.

Until this point, all results used digital neurons connected by memristors. Kulkarni and Teuscher demonstrated learning that does not involve neurons in the reservoir, and instead relied solely on a collection of memristors arranged in a random graph [21]. Through this model, Kulkarni and Teuscher were able to match the work by Pershin and Di Ventra and achieve associative learning. This is a significant boost to atomic switches: manufacturing networks of atomic switches with neurons in the junctions is more difficult than manufacturing a network without them.

Work has continued on the practical features of memristor networks, and has presented important results. Networks of memristors are very tolerant to variations [4], unlike traditional computing hardware which requires a strict adherence to device tolerances, else there could be irrecoverable failure. Additionally, more complex networks consisting of a "reservoir of reservoirs" is possible, and can potentially perform better than a single reservoir [5]. Progress on how to simulate these networks has also moved forward, including work by Konkoli and Wendin [18] and Smith [38], resulting in fast simulations of up to 100 memristors using Kirchhoff's Laws. There is also some work on determining the quality of a reservoir [19, 22], however this work is still in its infancy, with limited consensus on what identifies a good reservoir.

3 Simulating Novel Hardware

"One man's constant is another man's variable."
— Alan Perlis

The first phase of this project was to simulate the physical implementation of the hardware. We start with a simulation to enable rapid prototyping and experimentation before moving on to hardware experiments. The initial goal of using the actual hardware was abandoned due to the severely limited read and write time resolution, which was in the order of a second. Thus although the initial design of the simulator was to allow substituting in the hardware with minimal changes, this hardware layer was never implemented.

3.1 Percolation networks

A *percolation network* is most easily considered as a repeating planar graph where each node connects to a neighbour with probability p [39]. In particular, for $p > \frac{2}{3}$, a given sequence of vertices can be connected as a path. Thus changing p changes the characteristics of this graph, and so changes the paths through the network.

Percolation networks are worthy of mention because they provide a model of the way that tin particles behave in the atomic switch network. So long as the coverage p_c is kept below the percolation threshold of $\frac{2}{3}$, the network does not form a path between the two terminals. Because the tin particles are not uniform in size, the percolation threshold is 0.676336, not an exact $\frac{2}{3}$ [9]. In either case, because of the probabilistic nature of the connections, networks below the percolation threshold can "short-circuit", while networks over the percolation threshold can still be disconnected and so form an atomic switch network.

The simulations by Fostner and Brown simulate individual tin particles, and so spend a large amount of time randomly depositing the particle centroids [9]. The centroids are assigned a radius, and then checked to see if any of the particles overlap. When particles overlap, the form what we will call a *group*, which acts as a single conducting unit. We assume that this single conductive unit has zero resistance, because although this is incorrect, the true resistance is orders of magnitude smaller than the memristors and atomic switches that also populate the network, and is thus negligible.

When considering how this approach works, it became clear that we did not need this level of detail in our simulations. In fact, we would quickly be ignoring most of the work because the basic unit we wish to deal with is a group, not a particle. Because the number of particles is an order of magnitude greater than the number of groups, it is wasteful to generate all of them to throw them out again so soon. Furthermore, we don't care how the individual particles arrange themselves, only the final characteristics of the groups they form.

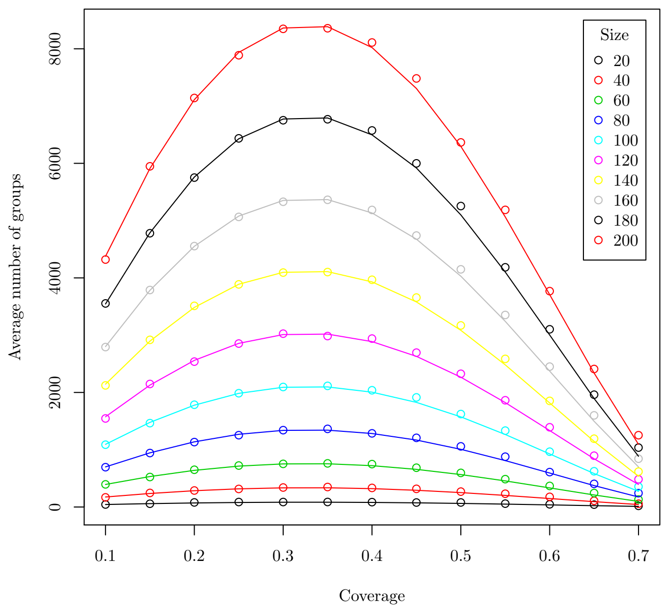


Figure 3.1: The average number of groups parameterised over coverage and chip side lengths, measured in particle radii and assuming a square chip (points), with the models (lines) used to approximate the number of groups.

Taking this into consideration, we consider the simulation from a higher level. Building on work by Fostner and Brown and the NRG, we use their existing simulations to build a dataset of boards from which we can extract key metrics. By repeatedly generating boards using certain parameters, we can explore how size and coverage influence the number of groups that will form, and the distances between these groups. The goal is to develop a probability distribution that yields numbers with the correct range of values for a board without the need to simulate particle deposition. This means we can move straight to placing whole groups, saving an order of magnitude in time even before considering the time saved by not needing to construct the groups from particles.

Using a fourth-degree polynomial, we can accurately model the number of groups over a wide variety of chip sizes and coverages, with R^2 coefficients of determination above 0.999. This accurate model means that a significant portion of information about the board is contained in five numbers, a_0 through a_4. The final model used is

$$g(p, x, y) = xy \cdot (0.0145 + 1.0274p - 0.4395p^2 - 3.7259p^3 + 3.2781p^4) \tag{3.1}$$

relating the width x, height y, and coverage p to the number of groups g. The g groups are then randomly deposited onto a simulated board. A comparison of the actual data and the models can be seen in Figure 3.1.

After depositing the groups, the connections between them need to be found. Because we do not have any particles to query about positioning, we need a new method. The groups must connect in a planar way, so we use a Delaunay Triangulation to determine the underlying graph. A Delaunay Triangulation will not generate parallel edges, a scenario possible in the physical structure. Because current will always favour the greatest conductance, and two resistors (a suitable model for the instantaneous state of memristors and switches) in parallel will act as a single resistor, we do not consider this limitation significant [38, Chapter 3.3].

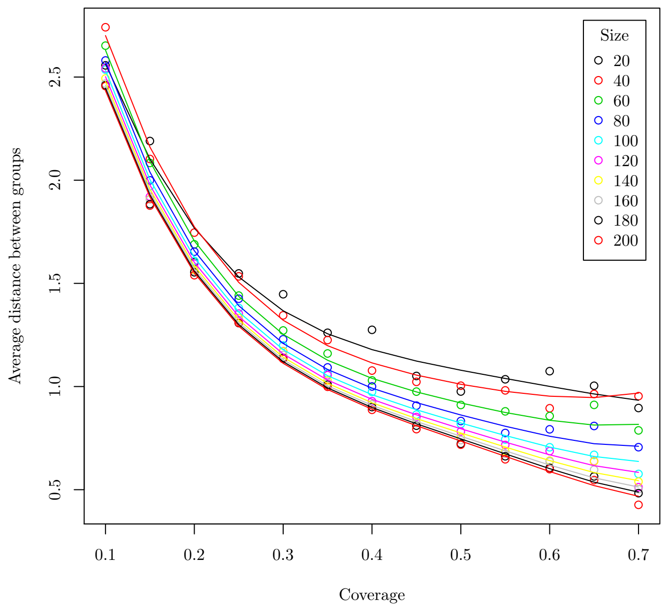


Figure 3.2: The mean distances between groups parameterised by coverages and chip side lengths, measured in particle radii and assuming a square chip (points), with the models (lines) used to approximate the mean distance.

In addition to finding the number of groups and how they connect, we must determine how far the groups are from one another. This is because the conductance of the *tunnels* between the groups is a function of the distance between the groups,

$$G(i,j) = \alpha e^{-\beta \ell(i,j)} \tag{3.2}$$

where $G(i,j)$ is the conductance of the gap between groups i and j of size $\ell(i,j)$ [9]. α and β are empirically derived parameters, typically set to be $\alpha = 1$ and $\beta = 100$. For this assignment, we use $\beta = 10$ to avoid potential numerical instability, as $e^{-100} \approx 3.7 \times 10^{-44}$. Although this is within the range of a double-precision floating point number, we make this adjustment because the resolution needed for this project is very fine, and because the change of β does not impact the result in any meaningful way beyond changing the magnitude of the starting conductances.

The distances ℓ themselves are unknown without the individual tin particles, as the groups we have defined above are infinitesimal centroids. To assign distances to the tunnels, we turn again to the simulations from Fostner and Brown. As before, by sampling distances from the existing simulations we can avoid simulating every particle, and instead model the distances with a distribution, which we can draw from instead.

We initially assumed that this distribution would be normal, so we calculated the mean and standard deviation for each grid size and coverage. When conducting simulations to determine if the generated networks matched the behaviour of the expected results, there was a significant discrepancy. When inspecting the mean, minimum, and maximum distances in the network it became clear that the maximum distance and minimum distance were not balanced around the mean value. This means the distribution has a significant skew which, once accounted for with a scaled beta distribution, yields accurate simulations. The final

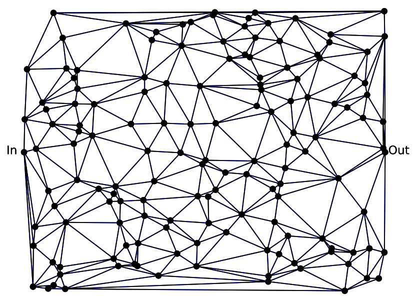


Figure 3.3: An example network generated using the group models, distance models, and a Delaunay Triangulation. Note that the edges are not proportional to the distances between the groups.

distances ℓ are drawn from

$$Xr + \epsilon \tag{3.3}$$

where

$$X \sim \text{Beta}\left(1, \frac{1-\mu}{\mu}\right) \qquad \text{and} \qquad \mu = \frac{\bar{\ell} - \epsilon}{r} \tag{3.4}$$

such that $\bar{\ell}$ is the mean distance for the given size, r is the range of values, and ϵ is the smallest distance between two groups before they would be considered a single group.

Empirical results suggest that the range is always between essentially 0 and 30 units, thus we set $r = 30$, and ϵ small, around 1×10^{-10}. We model $\bar{\ell}$ in a similar manner to the number of groups, again using a polynomial and consistently reach R^2 values above 0.97. But the model is less “clean”, requiring a deeper level of modelling than was suitable for the number of groups. For a given board size $x \times y$ with coverage p, we can write the average distance between groups on the board $\bar{\ell}$ as

$$\bar{\ell}(x, y, p) = a(p) + \frac{b(p)}{\sqrt{xy}} + \frac{c(p)}{xy} \tag{3.5}$$

where

$$\begin{aligned} a(p) &= 3.90 - 20.76p + 55.78p^2 - 71.48p^3 + 33.67p^4 \\ b(p) &= 39.28 - 172.59p + 437.66p^2 - 497.85p^3 + 334.98p^4 \\ c(p) &= -749.77 + 4405.25p - 12599.52p^2 + 16937.46p^3 - 10645.31p^4. \end{aligned} \tag{3.6}$$

The $\sqrt{xy}$ term comes in because it represents the geometric average of the side lengths, and so this polynomial can be viewed as a quadratic function of the inverse of the geometric mean board side length. The models can be seen in comparison to the actual data in Figure 3.2. Combined with equation (3.1), we can approximate everything about the board using only x, y, and p.

By combining both the group models, the distance models, and a simple Delaunay triangulation, we are able to simulate the result of tin particle deposition in a percolation network for a variety of board sizes, with samples from 20×20 through to 200×200 units, and coverages p varying from 0.1 through to 0.7. We use this model moving forward with the simulations of circuits composed of these groups and connections. We will subsequently refer to this board simulation as the *chip*. An example of the final product of these models can be seen in Figure 3.3, although the edges between vertices are not proportional to the distance between the groups.

3.2 Kirchhoff's laws

Although we deviated from previous work in constructing the simulated network, the underlying calculations are essentially the same. Hence most of the following is a restatement of the work by Smith [38]. We introduce multiple input connections with distinct voltages as a minor extension over the existing work.

Given a circuit, how does one simulate the current and voltage passing through it? At the core of the answer lie two fundamental physical equations, known collectively as *Kirchhoff's Laws* [36, Section 28.2]. The first, Kirchhoff's Current Law, is defined in the following Definition 3.1. The second law is Kirchhoff's Voltage Law, defined as follows in Definition 3.2.

Definition 3.1 (Kirchhoff's Current Law)
The directed sum of current at a vertex must be zero. That is,

$$\sum_{j \in \mathcal{N}(i)} I_{ij} = 0 \tag{3.7}$$

where $\mathcal{N}(i)$ is the set of all neighbours of vertex i, and I_{ij} is the current between vertex i and vertex j, signed such that currents flowing from j into i are negative.

Definition 3.2 (Kirchhoff's Voltage Law)
The directed voltage around a cycle must sum to zero.

Because of Definition 3.2, it is immediately clear that if the chip is the sole element in the circuit aside from the power source, then the output voltage of the network must be zero. Similarly, it becomes clear that an arbitrary voltage can be written to each input by assuming a resistor of appropriate value is in series with the input. Variable resistors make such an input simple to generate. By these results, we can focus all further attention upon the chip using the assumptions of arbitrary input voltages and zero output voltages.

The current in Equation (3.7) between every vertex is not trivially found, and so by using Equation (2.12) for memristors, or Ohm's Law for atomic switches, we can instead rewrite it as

$$\sum_{j \in \mathcal{N}(i)} G_{ij}(V_j - V_i) = \sum_{j \in \mathcal{N}(i)} G_{ij} V_j - V_i \sum_{j \in \mathcal{N}(i)} G_{ij} = 0. \tag{3.8}$$

We use the difference of voltages between two vertices because the voltage across the edge component is the voltage drop between the two vertices. The conductance between each node is initially set by Equation (3.2), and then updated based on the update rules for the components.

Importantly, this definition is valid only if the vertex is inside the network. If instead a vertex is in contact with an input or output connection, we must account for the extra voltage and current connection. To do this, we introduce I_{in} and I_{out}, two vectors which are nonzero in the entries that are in contact with the input and output connection, respectively. Thus we can now rewrite Equation (3.8) as

$$(I_{\text{in}})_i - (I_{\text{out}})_i + \sum_{j \in \mathcal{N}(i)} G_{ij} V_j - V_i \sum_{j \in \mathcal{N}(i)} G_{ij} = 0. \tag{3.9}$$

Finally the boundary conditions are trivially set using earlier assumptions. We set the voltage at the input connections to be the input voltage for that connection, and the output voltage is always zero.

Using Equation (3.9) and the boundary voltage conditions, we can create the matrix $\mathbf{G}$ and boundary vector $\mathbf{v}$, shown in Figure 3.4. The matrix $\mathbf{G}$ is constructed in a consistent way:

$$\underset{\mathbf{G}}{\left[\begin{array}{ccccc:c:c}
-(\sum G_{1j}) & G_{12} & \cdots & & & 1 & \\
G_{21} & -(\sum G_{2j}) & \cdots & & & \ddots & \\
\vdots & \vdots & \ddots & & & 1 & \\
 & & & & & & -1 \\
 & & & & & & \ddots \\
 & & & & & & -1 \\
\hdashline
1 & & & & & & \\
 & \ddots & & & & 0 & 0 \\
 & & 1 & & & & \\
\hdashline
 & & & 1 & & & \\
 & & & & \ddots & 0 & 0 \\
 & & & & 1 & &
\end{array}\right]}
\underset{\mathbf{x}}{\left[\begin{array}{c} V_1 \\ V_2 \\ \vdots \\ \\ \\ \\ I_1^{\text{in}} \\ I_2^{\text{in}} \\ \vdots \\ I_1^{\text{out}} \\ I_2^{\text{out}} \\ \vdots \end{array}\right]}
=
\underset{\mathbf{v}}{\left[\begin{array}{c} 0 \\ \vdots \\ \\ \\ \vdots \\ 0 \\ V_1 \\ V_2 \\ \vdots \\ 0 \\ \vdots \\ 0 \end{array}\right]}$$

Figure 3.4: Matrix $\mathbf{G}$, boundary vector $\mathbf{v}$, and associated linear equation to solve Kirchhoff's Laws for the voltages at every node.

1. Designate each of the g vertices as internal, input $\in \mathcal{I}$, or output $\in \mathcal{O}$.
2. Construct a square zero-matrix with size $g + |\mathcal{I}| + |\mathcal{O}|$ in each dimension.
3. In the top left $g \times g$ sub-matrix, set the elements as follows:
$$\mathbf{G}_{ij} = \begin{cases} -\sum_{k \in \mathcal{N}(i)} G_{ik} & \text{if } i = j \\ G_{ij} & \text{otherwise} \end{cases} \tag{3.10}$$
4. The $g \times |\mathcal{I}|$ sub-matrix horizontally adjacent contains a 1 in every entry $\mathbf{G}_{i,g+j}$ such that vertex i is attached to input connection j. The transpose is also filled in such a manner.
5. The $g \times |\mathcal{O}|$ sub-matrix horizontally adjacent again contains a -1 in every entry $\mathbf{G}_{i,g+|\mathcal{I}|+j}$ such that vertex i is attached to output connection j. The transpose is also filled in such a manner, except with 1 instead of -1.
6. The lower right $(|\mathcal{I}| + |\mathcal{O}|) \times (|\mathcal{I}| + |\mathcal{O}|)$ sub-matrix remains filled with zeros.
7. The boundary vector $\mathbf{v}$ is initially set to all zero, and has size $(g + |\mathcal{I}| + |\mathcal{O}|) \times 1$.
8. Fill $\mathbf{v}_{g+i} = u_i(t)$ where $u_i(t)$ is the ith input value at time t.

By solving the system of linear equations

$$\mathbf{Gx} = \mathbf{v}, \tag{3.11}$$

we are able to determine the voltage at every vertex, the current across the chip, and hence the current across each edge. Using the SciPy scientific library, we can solve the system of equations in Figure 3.4 using an LU decomposition through `scipy.linalg.solve`, running in $O(n^3)$ time where $\mathbf{G}$ is $n \times n$. Underneath, this makes a call to the LAPACK libraries, a well-tested library of matrix algebra functions.

3.3 Tunnels

The edges between vertices in the underlying graph structure of the chip have a physical interpretation in terms of groups. They serve as places for the electrons to pass from group to group, and so we refer to them as *tunnels*. For the purposes of this project, we consider a tunnel to be either a resistor, memristor, or an atomic switch. There is no technical reason why we are limited to these tunnels, and more exotic tunnel types will change the learning behaviour of the network.

All the tunnels follow a consistent interface, constructed based on their initial conductance and gap sizes, and updated based on the current and voltage passed through them. This means any class that follows this interface can be inserted into the chip. This enabled rapid prototyping and comparisons.

The tunnel calculations are applied to every tunnel concurrently. By collecting every tunnel into a matrix, we can perform operations using vectorised SciPy matrix operations, implemented in C or Fortran. This ensures that we keep the expressive—and fast to code—Python layer at the top of the stack, but use an appropriately efficient language at the numerically intensive layers.

Resistors are the simplest tunnel, acting as a linear transformation. They do not update because they are stateless, and so are simple to simulate in large quantities. They form a "sanity check" for the suitability of the simulator, because their behaviour is simple to predict and verify. The resistors would also form a baseline for what behaviour can be attributed to the chip, and what can be attributed to the reservoir learner's readout layer, discussed in Chapter 4.

A memristor is a more complicated form of tunnel. The conductance is a function of past inputs, and given by a differential equation. Solving this differential equation is nontrivial, because the memristor is not in isolation. If the memristor were in isolation, it would be a simple matter to approximate a numerical solution and consider it solved. Because we are dealing with a network of memristors, the solution is dependent not only on the past input, but also the past input of its neighbours. Although this makes simulations difficult, it does offer promise that these will behave much like the ESNs that we are using to model the learning side of this hardware.

To overcome the difficulties associated with solving this differential equation, we work back to the most basic definition. By taking a 'single' time step and slicing it finely, we iterate for a sufficiently good approximation, and thus end up with an Euler discretisation:

$$\frac{dx}{dt} = f(x) \rightsquigarrow \Delta x = f(x)\Delta t. \tag{3.12}$$

Hence we set Δt small and iterate towards a solution. This approach is unfortunate in that it does increase the time complexity by a factor of $k = 1/\Delta t$, but it does yield sufficiently accurate results. In an attempt to extract every bit of performance out of the simulator, the memristor update procedure is written in Fortran.

The final important tunnel type used in this project is the atomic switch. Their modelling is more straightforward than that of the memristors, because there is no "transition"—a switch is either off (low conductance), or on (high conductance). The low conductance state is as defined in Equation (3.2), and the high conductance is set at $10\,\Omega^{-1}$. There are also switching conditions based on the electric field induced through a tunnel, and the current passing through a tunnel. Because of the random nature of the tin particles, there is a probability parameter ($P_\uparrow$ and $P_\downarrow$) controlling each switch direction [9].

A tunnel will switch from the low conductance state to the high conductance state under two conditions: first, a random uniform variable P satisfies $P < P_\uparrow$; second, the field across the tunnel is above a threshold field strength E_T. The field strength E across a tunnel of length ℓ is given by

$$E = \frac{\Delta V}{\ell} \tag{3.13}$$

where ΔV represents the voltage drop across the tunnel. This is a more accurate model than the voltage threshold model from Equation (2.15), because the forces induced by the field cause the tin particles to move and form the "bridges" causing the high conductance.

The switch down condition is different to the switch up condition. Again, we draw a uniform variable P to satisfy $P < P_\downarrow$, but no longer consider the field strength across a tunnel. A bridge between two groups will break should the current across the tunnel exceed a threshold current I_T. The switch down probability is typically set substantially lower than the switch up probability, because it is more difficult to break a bridge than to form it. Fostner and Brown showed that switch down conditions only add noise, and do not affect the overall results [9].

Regardless of tunnel type, it is important to be able to determine what is happening inside the network. Although in simulations it is trivial to measure every tunnel, on the physical chip this is more challenging. After consultation with Professor Brown, we have determined that a matrix of sensors is a viable method to sample the network. Thus our simulations do not return the tunnel measurements, but average over an area of the chip and feed into a sensor. The chip is divided into a grid, and each tunnel is assigned to a grid position based on the midpoint of the edge between two group centroids. The mean of the currents through each tunnel that is in a grid position forms a single datapoint in the readout. This means we have information about the current from every part of the chip, much like we would have in a traditional ESN.

3.4 Putting it together

The concept of a tunnel has so far been kept distinct from the concept of a chip. Because of the use of consistent interfaces within the simulation, any class that adheres to the `Tunnel` protocol is suitable to serve as a tunnel, meaning that we can model a wide variety of chips with the same chip class, which we call `MemChip`. Figure 3.5 serves as a structural guide as to how all this fits together as a UML diagram. The private methods that go in to generating the simulated chip, such as the random distributions and chip layout, are not exposed nor included in the UML diagram.

Keeping in mind the code was initially designed to wrap around actual hardware, the interface is sparse. A `MemChip` is initialised at a certain size, coverage, and tunnel class, with other optional parameters. The optional parameters are the number of inputs and outputs, whether to use the sensor grid, and an `override_depositions` flag to use a specific chip structure. There are two other ways to initialise a `MemChip`. The first is `MemChip.with_groups`, which replaces the width, height, and coverage parameters with a single groups parameter, which specifies how many groups should be on the chip. A square `MemChip` is then generated with coverage 0.65 with appropriate sidelength. The second alternative way to generate

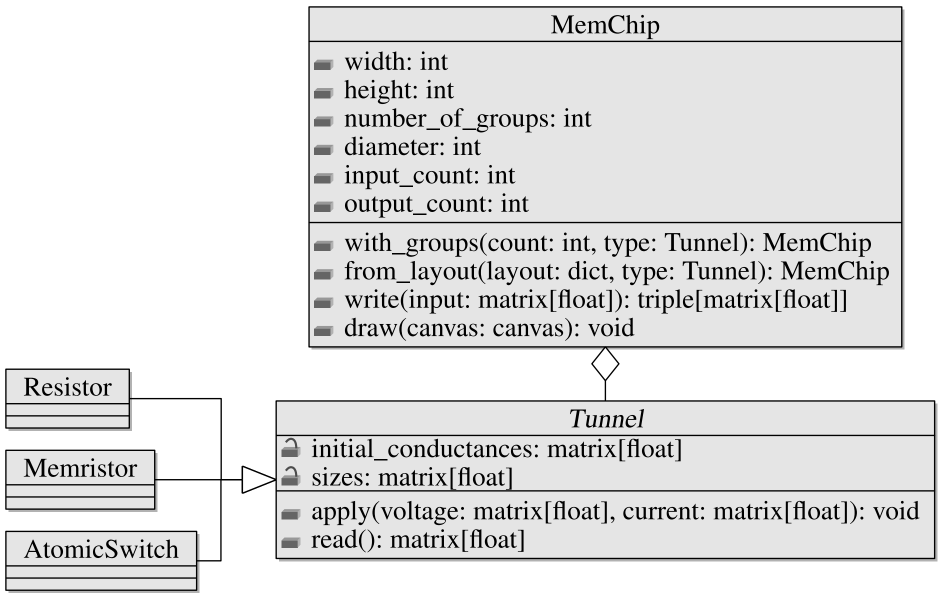

MemChip
width: int
height: int
number_of_groups: int
diameter: int
input_count: int
output_count: int
with_groups(count: int, type: Tunnel): MemChip
from_layout(layout: dict, type: Tunnel): MemChip
write(input: matrix[float]): triple[matrix[float]]
draw(canvas: canvas): void
Resistor
Memristor
AtomicSwitch
Tunnel
initial_conductances: matrix[float]
sizes: matrix[float]
apply(voltage: matrix[float], current: matrix[float]): void
read(): matrix[float]

Figure 3.5: A UML diagram of the `MemChip` and `Tunnel` classes.

a `MemChip` is with `MemChip.from_layout`. With this initialisation, the width, height, and coverage parameters are replaced with a chip structure, specified as

```
Dict {
  node_index: (
    (x_location, y_location),
    [(neighbour_index, distance) for each neighbour])}
```

which can be used for more unusual or specific chip layouts, such as those required by Pershin and Di Ventra and their maze-solving [29].

Once the chip has been initialised, it can be drawn using the `draw` method, which accepts a canvas to draw on, and two parameter which control what is presented. The first parameter is a flag controlling drawing the underlying grid structure, while the second optional parameter is the conductance of the tunnels at a given time. The second parameter thus implicitly controls the opacity of the edges, meaning that a high conductance is more opaque, while a low conductance is more transparent. The key reason for drawing these networks is to understand how they assemble themselves. Because these networks are statistically generated, we were unsure in advance the actual structure these chips could take. By drawing them, we are able to reason more effectively about the interaction between the groups and discuss the physical properties they might have.

A chip can also be "written to", meaning to apply a sequence of voltages to the input groups, using the `write` method. The input format is as a matrix where each row t is an individual input vector $\big(\mathbf{u}(t)\big)^\top$. The matrix will have τ rows, one for every input time. At each time t, the input vector $\mathbf{u}(t)$ is applied to the chip and output data collected. This output data is constructed into corresponding matrices where row t contains the output at time t.

The chip collects three kinds of outputs, and returns them as a triple. The first element of the triple is a vector of the conductance of the entire chip for the corresponding input. The conductance of the entire chip at time t is

$$G(t) = \frac{\sum_{i \in \mathcal{I}} |I_i|}{|u_0(t)|} \tag{3.14}$$

where $u_0(t)$ is the first element of the input-voltages vector $\mathbf{u}(t)$. This makes the assumption that every entry in the input-voltages vector is the same, and if this is not the case the question of the conductance of the chip has no strict interpretation, and thus no sensible answer.[1] Thus the first element of the triple is `None` in the case of multiple input voltages. The second element of the triple is the matrix of currents passing through the output groups, where row t contains the currents coming out at time t. The final element of the triple is the matrix of sensor grid readings, each time step represented as a row, and each sensor's reading is an element in that row. From these three types of outputs we should be able to extract any potential learning information in the chip.

Now the chip can be created, written to, and read from. The time complexity of creating the chip is dominated by the Delaunay Triangulation, which runs in $O(n \log n)$ time, where n is the number of groups. Reading to and writing from the chip are intertwined, and their complexity is a product of the resolution of the differential equation solver, and solving the linear equations. Together, the time complexity of the chip simulation is $O(kn^3)$, where n is again the number of groups and $k = 1/\Delta t$ is the number of iterations needed to solve the differential equation in Equation (2.14). The number of iterations is inversely proportional to time estimated with each differential equation iteration, Δt.

Because of the unavoidable complexity involved with this simulation, in Python the results were unacceptably slow. To overcome this, we decided to move the slowest parts of the logic to Fortran. Fortran was chosen for two main reasons: first, the simulation works extensively in matrices and linear algebra, essentially the raison d'être of Fortran; second, the `f2py` utility from the SciPy project makes combining Fortran and Python code trivial. Modern Fortran is highly readable, exceptionally fast, and compatible with OpenMP, a library that enables simple parallelisation, the kind possible with the matrix operations we perform.

[1]Equation (3.14) is essentially $G = I/V$, using a special case of I and V for our chip. Multiple input currents is simple to account for because total current is the sum of parallel currents. Multiple voltages is less obvious, because voltage does not add this way, and instead of being conserved like current it must be consumed on its path to the sink. From this there are two concerns: where does the path lead, and what is the conductance along this path. The first concern comes from the fact that if the energy drop between two inputs is high enough, and the resistance between them low enough, the current will actually run from one input to the other. This can be worked around with diodes, but in general illustrates an issue with asking about the conductance of a multiple-input device like the chip. The second concern leads to the conclusion that each input produces a separate conductance measurement for the chip, because the path it takes *will* have a different conductance. This means the "conductance" of the chip is actually not a single number at all, but a matrix of measurements relating every terminal (both input and output) to every other terminal. However, actually calculating this matrix is difficult because the paths are not independent, and is well outside the scope of this project.

4 Constructing a Reservoir

"All models are wrong, but some are useful."
— George Box

Reservoir computing as a paradigm is well-suited to hardware implementations due to the fixed nature of the weights between neurons in the reservoir itself. For this reason it became the basis of learning in this project. In this chapter we discuss our reservoir computing implementation, and how we integrated the hardware simulations into the learner, as well as some of the reservoir features associated with learning.

4.1 Abstraction

A reservoir learner is essentially a composition of three layers: an input layer, the reservoir, and a readout (or output) layer. Each layer is worth considering individually, because it becomes clear that they can extrapolate out to interfaces that, once implemented appropriately, become modular and useful.

We consider first the input layer. This layer is potentially the most basic. We define the interface as the transformation $input : \mathbb{R}^l \to \mathbb{R}^m$ for arbitrary l and m. These minimal restrictions mean that essentially arbitrary transformations are possible. But this does not mean that arbitrary transformations are useful. For this project, we focus on two transformations: *identity*, and *bias*. The first, *identity*, is as it sounds—the identity transformation. The point of this transform is to enable us to act as if there was no input filter, and carry on anyway. The second is more important, because *bias* is an important and necessary part of machine learning. This transform is defined by the simple mapping

$$\mathbf{x} \mapsto [1; \mathbf{x}]. \tag{4.1}$$

Bias in this particular form is also part of the definition of an ESN.

The readout layer is a more complex layer, and is the layer that actually partakes in the "learning" as it were. This means the layer must be sufficiently powerful to train and map reservoir-transformed data to desired outputs, but should also ideally be easy to train—if the readout layer is not easy to train, a key benefit of reservoir computing disappears. The interface is again broad, but now consisting of two functions. The first is training: $train : \mathbb{R}^{\tau \times n} \times \mathbb{R}^{\tau \times o} \to \varnothing$, working on an entire matrix of outputs as may be required by the learning algorithm—τ is the number of training examples. Although a multitude viable options exist such as genetic algorithms [21], the most common is a variant of linear regression. The version we chose for this research is *ridge regression*, which will be covered more in Section 4.2. An important consideration is the implicit learning power of the readout layer without any influence from the reservoir. Many of the readout algorithms are themselves

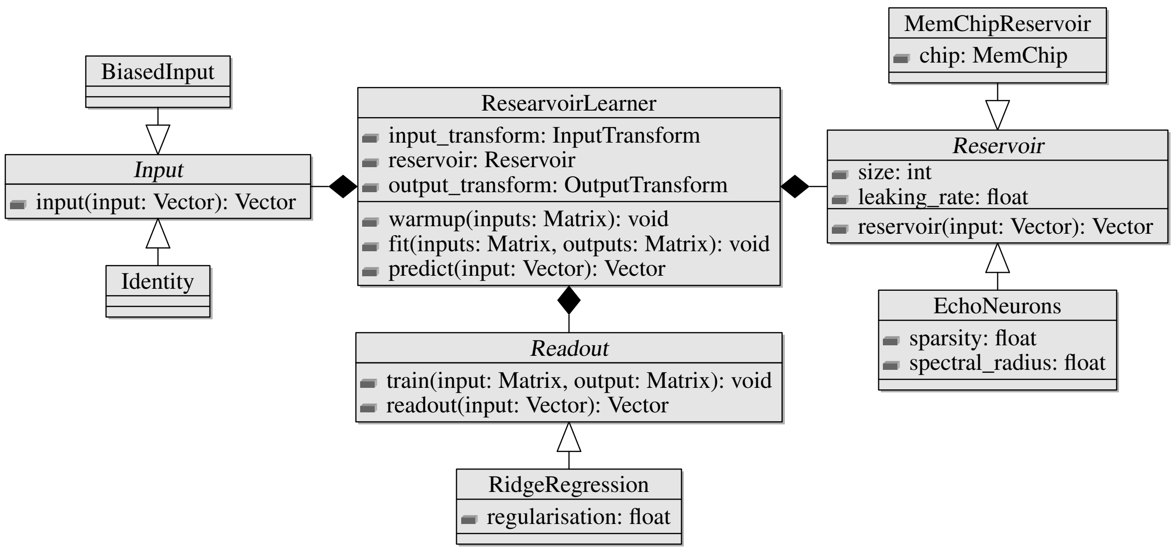

BiasedInput
Input
input(input: Vector): Vector
Identity
ResearvoirLearner
input_transform: InputTransform
reservoir: Reservoir
output_transform: OutputTransform
warmup(inputs: Matrix): void
fit(inputs: Matrix, outputs: Matrix): void
predict(input: Vector): Vector
Readout
train(input: Matrix, output: Matrix): void
readout(input: Vector): Vector
RidgeRegression
regularisation: float
MemChipReservoir
chip: MemChip
Reservoir
size: int
leaking_rate: float
reservoir(input: Vector): Vector
EchoNeurons
sparsity: float
spectral_radius: float

Figure 4.1: A UML diagram of a reservoir learner.

capable learners, and care must be taken when attributing learning. The second method exposed by the readout layer is *readout* : $\mathbb{R}^n \to \mathbb{R}^o$, which performs the prediction as learned via *train*.

The reservoir is the most variable layer, with an interface again flexible: *reservoir* : $\mathbb{R}^m \to \mathbb{R}^n$. To fully meet the definition of an ESN, the reservoir should also satisfy the echo property—that is, the current state should be an injective function on the entire history. In this research, the reservoir received the most attention, being where the memristive hardware of interest becomes relevant. We also define reservoirs matching the specification of an ESN, as well as variations of ESNs. Further variation of reservoirs is possible, and an avenue for future research.

Together, these three layers can form a complete reservoir learner. Figure 4.1 contains a UML diagram outlining the structure of the reservoir learner. The reservoir learner exposes methods corresponding to three distinct steps: warm-up, fitting, and prediction. The fitting workflow is straight forward: for each input vector $\mathbf{u}(t)$, calculate the transformed vector

$$\mathbf{x}(t) = (\mathit{reservoir} \circ \mathit{input})(\mathbf{u}(t)); \tag{4.2}$$

collect these transformed vectors together as a matrix $\mathbf{X}$ such that each row is $\mathbf{x}(t)^\top$; train using the matrix $\mathbf{X}$ against the expected output matrix $\mathbf{Y}^{\text{target}}$ via $\mathit{train}(\mathbf{X}, \mathbf{Y}^{\text{target}})$. $\mathbf{Y}^{\text{target}}$ consists of rows of the expected output vectors $\mathbf{y}(t)^\top$. The warmup phase is a simple step in which data is fed to the network much like in the fitting phase, but makes no attempt at training. Finally, prediction is the composition

$$\mathbf{y}(t) = \mathit{output}(\mathbf{x}(t)) = (\mathit{output} \circ \mathit{reservoir} \circ \mathit{input})(\mathbf{u}(t)). \tag{4.3}$$

In practice it is not this tidy, as often the input vector $\mathbf{u}(t)$ is vertically concatenated onto the vector $\mathbf{x}(t)$ before being sent to the output layer, however this can be considered a function of the reservoir and thus preserving the data flow as defined above.

As an illustrative example, we outline how a “standard” ESN could be implemented in this model. Consider first Equations (2.7) defining an ESN, notably the inclusion of a bias added to the input $\mathbf{u}(t)$. Thus the input layer is BiasedInput, using the function from (4.1). We call the result of the input $\mathbf{v}(t) = [1; \mathbf{u}(t)]$. The reservoir layer of the ESN is the most complex. Building atop Equations (2.7) we can write

$$\begin{aligned}\mathbf{x}(t) &= [\mathbf{v}(t); \mathbf{x}'(t)] \\ \mathbf{x}'(t) &= (1-\alpha) \times \mathbf{x}'(t-1) + \alpha \times \tanh\left(\mathbf{W}^{\text{in}}\mathbf{v}(t) + \mathbf{W}\mathbf{x}'(t-1)\right).\end{aligned} \tag{4.4}$$

Both $\mathbf{W}$ and $\mathbf{W}^{\text{in}}$ are simply random matrices with entries drawn from a uniform random distribution over the range $[-0.5, 0.5]$, made sufficently sparse, and then scaled according to the desired spectral radius. Finally, the readout layer is a linear ridge regression, which will be outlined in more detail below. Briefly, we can write

$$\mathbf{y}(t) = \mathit{output}(\mathbf{x}(t)) = \mathbf{W}^{\text{out}}\mathbf{x}(t) \tag{4.5}$$

assuming an already trained $\mathbf{W}^{\text{out}}$.

The main focus of this research is of course having the `MemChip` serve as the reservoir. To accomplish this, we create a small wrapper around a `MemChip`, ensuring it correctly implements the *reservoir* interface. The input layer is usually taken to be identity, but this is not a requirement. The readout layer is a linear map trained via ridge regression, like in the ESN implementation. We take the output vector of the reservoir to be the current readings from the sensor grid.

4.2 Readout weights

Now that we have established a framework for reservoir learning, we turn our attention to the readout layer as implemented in this project. As mentioned earlier, a wide variety of approaches are viable, but we chose to implement a linear regression using ridge regression. This decision was made because linear regression is a sufficiently powerful learning method to extract the necessary information from the reservoir, while still being simple to train, and not so expressive that it would be capable of the entirety of the learning.

The readout layer, being a linear regression, is very simple to use once trained—see Equation (4.5). Thus the only remaining problem is how to determine the values of $\mathbf{W}^{\text{out}}$. Again, many possible solutions to this problem exist, and it is well-studied already. Options include gradient descent, direct and pseudoinverse calculations, and—our chosen approach—ridge regression, also known as Tikhonov regression [24]. Ridge regression is an advancement on what are commonly known as the *normal equations*, adding a regularisation coefficient β, which serves as a penalty against large values in $\mathbf{W}^{\text{out}}$.

When attempting to “solve” for $\mathbf{W}^{\text{out}}$, the underlying goal is best stated as finding some matrix that minimises the distance between its approximation and the true values it attempts to learn, all while penalising unusually high entires. Compactly,

$$\mathbf{W}^{\text{out}} = \underset{\mathbf{W}^{\text{out}}}{\operatorname{argmin}} \frac{1}{|\mathbf{y}|} \sum_{i=1}^{|\mathbf{y}|} \sum_{t=1}^{\tau} (y_i^{\text{target}}(t) - y_i(t))^2 + \beta \|\mathbf{w}_i^{\text{out}}\|_2 \tag{4.6}$$

where $\mathbf{w}_i^{\text{out}}$ is the ith row of $\mathbf{W}^{\text{out}}$ [24], $|\mathbf{y}|$ is the number of elements in the vector $\mathbf{y}$, and $\|\cdot\|_2$ is the length taken as the Euclidean norm of a vector. This minimisation can be condensed down to the closed-form equation

$$\mathbf{W}^{\text{out}} = \mathbf{Y}^{\text{target}}\mathbf{X}^\top \left(\mathbf{X}\mathbf{X}^\top + \beta\mathbf{I}\right)^{-1}. \tag{4.7}$$

Thus the learning happens in one step once all the training examples have been provided.

4.3 Testing and configuration

The ideas around what makes a good reservoir are varied, with little agreement on what makes one better than another. A common and promising approach is harmonic generation, but this is difficult to quantify and has little to no reference point. Instead, we will focus on statistical tests to determine how reservoirs relate to one another. For this, we will need some data to learn. We have chosen two data sets: one time-independent, the other time-dependent. Together, these should give us a better understanding of what the networks can learn. Additionally we will be testing a range of reservoir learners. Along side the ESN and atomic switch network, we will have a memristor network and a resistor network.

4.3.1 Datasets

The Mixed National Institute of Standards and Technology (MNIST) database is a standard machine learning dataset. Composed of over 70 000 images, the MNIST database contains hand-written digits (0–9) converted to greyscale and scaled to 20×20 pixels. This dataset is extensively studied, so we can know in advance how the performance of our classifier compares to others. Note that this dataset has no time dependency. Because all our learners should have memory, to prevent them learning by counting, we shuffle the dataset.

In comparison to the MNIST database, we needed a dataset that required the learner to remember the past to predict the future. Many such datasets exist, such as sunspots or stock markets, but the dataset we have chosen is the Mackey-Glass series. The Mackey-Glass series is attractive because it is defined in terms of a differential equation with a "difficulty" parameter τ, ranging from 17 upwards, with the complexity of the curve increasing with τ. The full form of the Mackey-Glass equation is

$$\frac{dx}{dt} = \beta \frac{x_\tau}{1 + x_\tau^n} - \gamma x, \quad \beta, \gamma, n > 0, \tag{4.8}$$

where β, γ, and n are all real numbers, and x_τ is the value of the x variable at time $t - \tau$. Thanks to τ, the difficulty of the problem can increase to explore the learning potential each of our neuromorphic learners have. The particular values used are in Appendix B.

4.3.2 Reservoirs

Every experiment needs a baseline, and the ESN serves this purpose as an upper reference. While it is unlikely that these learners will exceed the performance of an ESN, it serves as a goal for which we can aim. The ESN is (relatively) well understood [24], and although it has many parameters to tune, the performance is largely decided by three key values: spectral radius, leaking, and regularisation. The values we used for each parameter of each learner are recored in Appendix B. Thus we can be confident the ESN we use for testing is sufficiently tailored to the challenge set to it. Worth noting is that an ESN was not designed for use with a time-independent dataset, but instead is designed for temporal datasets. This makes the MNIST database an unfair test, but it does reveal what impact the reservoir and readout layer have on learning.

The memristor network is the first neuromorphic reservoir that we tested. Consisting of a model identical to the atomic switch network but with memristors in the place of atomic switches, this network was initially going to serve as a comparison against the work of others, to see how the simulator outlined above performs in relation to existing memristor simulations. However, because the network in the simulations we ran is homogeneous, there are limited parallels with existing work—almost all research has chosen to focus on reservoirs with digital

$$\begin{tikzcd}
 & & \text{ESN} & \\
 & \text{AESN} \arrow[ur] & \text{NESN} \arrow[u] & \text{CESN} \arrow[ul] \\
 & \text{FFESN} \arrow[u] \arrow[ur] & \text{CAESN} \arrow[ul] \arrow[ur] & \text{CNESN} \arrow[u] \arrow[ul] \\
\text{OFFESN} \arrow[ur] & & \text{CFFESN} \arrow[ul] \arrow[u] \arrow[ur] & \\
 & \text{OCFFESN} \arrow[ul] \arrow[ur] & &
\end{tikzcd}$$

Prefixes

A	Acyclic
C	Conservative
FF	Feed-forward
N	No-echo
O	One-hop

Figure 4.2: A diagram of reservoirs and how they relate. Arrows represent the "is a restriction of" relationship.

neurons alongside the memristive hardware. As with the ESN, the parameters we use are in Appendix B. Although the hardware we are simulating is restricted to a single input and output, we adjust this number as necessary for our learners because this can change in hardware.

Atomic switch networks represent the hardware built by the Nanotechnology Research Group at the University of Canterbury, constructed as outlined in Chapter 3. Atomic switches can be considered a type of memristive hardware, despite being constructed entirely differently. Although they bear a resemblance to memristors, the instantaneous and binary nature of their conductance may present new learning potential or restrictions. We wrap the chip simulation in a simple wrapping class meaning that it can expose the correct reservoir interface, and can insert it with minimal difficulty into a learner much like an ESN.

The last neuromorphic hardware chip we simulate is a network of resistors. This network serves as the lower baseline of how these reservoirs should perform. A network of resistors behaves exactly as a single large resistor, so there is no potential for any learning outside what the readout layer is capable of. Resistors also enabled us to test exactly that—the capabilities of the readout layer. This means we know where to attribute the learning that we observe in the network.

When running these tests, it became clear that there was a steep disparity between the ESN and the neuromorphic reservoir simulations. To explore why this was, we ended up creating a wide variety of reservoirs, each with or without certain features present in an ESN or neuromorphic reservoir. However, some features of a reservoir preclude others. Because of this, a family of reservoirs was developed, and the inheritance structure is complex. The total collection of reservoirs is shown in Figure 4.2. The reservoirs alter a pipeline available to change the structure of the reservoir in a consistent way, and thus build up more complex combinations of feature restrictions. The exact choice of which features to remove is discussed in Section 5.3, along with how they are related and which features are removed. We consider an "unweakened" ESN to be the most powerful version, while the "one-hop" ESN represents the weakest form of learner. We wish to determine where a memristive neuromorphic chip belongs in this taxonomy.

4.3.3 Testing

Because learning is not a guaranteed process, we allow each learner to have ten attempts at learning the data with a different random seed each time. Each time the learner did not fail to learn we evaluate how it performed. The exact method of evaluation differs between the MNIST database and the Mackey-Glass predictions because of the nature of the datasets.

We draw attention to the fact that “did not fail to learn” is not synonymous with “successfully learned.” Precisely, we consider “did not fail to learn” to mean that the learner produced a bounded model of the dataset, but place no requirement that this model accurately predicted the data.

When testing learners on the MNIST database, we provide them with first half of the database as training data, and the second half as evaluation data. We present the individual (per-digit) and overall (mean) precision and recall of the learner over the dataset, both for each individual learner of the ten and the mean value. Using the notation that $\mathrm{pred}(c)$ is the set of instances predicted to be of class c and $\mathrm{act}(c)$ is the set of instances truly in class c, the precision of a learner on class c is defined as

$$\mathrm{precision}(c) = \frac{|\mathrm{pred}(c) \cap \mathrm{act}(c)|}{|\mathrm{pred}(c)|} \tag{4.9}$$

and the recall of a learner on class c is

$$\mathrm{recall}(c) = \frac{|\mathrm{pred}(c) \cap \mathrm{act}(c)|}{|\mathrm{act}(c)|}. \tag{4.10}$$

A slightly different approach is taken for the Mackey-Glass prediction test. Because there is not a finite number of classifiers, we must use a measure of accuracy that is more suited to the desired outcome. In this case, we are interested in how closely the predicted curve follows the actual curve. For this, we use a metric called the *correlation distance*, measuring the distance between two curves represented by vectors $\mathbf{a}$ and $\mathbf{b}$ by

$$d_{\mathrm{corr}}(\mathbf{a}, \mathbf{b}) = 1 - \frac{(\mathbf{a} - \bar{\mathbf{a}}) \cdot (\mathbf{b} - \bar{\mathbf{b}})}{\|\mathbf{a} - \bar{\mathbf{a}}\|_2 \|\mathbf{b} - \bar{\mathbf{b}}\|_2}. \tag{4.11}$$

The symbol $\bar{\mathbf{a}}$ is the mean value of $\mathbf{a}$ with subtraction applied element-wise, $\cdot$ is the dot product of vectors, and $\|\cdot\|_2$ is the Euclidean norm. This particular distance metric was chosen as it better captures the intent to follow a curve rather than how far apart two curves happen to be. Using this metric, we are able to explore more thoroughly how two learners perform relative to one another when attempting time-series prediction.

5 Comparisons and Results

"A big computer, a complex algorithm and a long time does not equal science."
— Robert Gentleman

Throughout this project, testing and evaluating has been an ongoing process. Different tests have driven development down different paths, and the results given in this chapter highlight the important milestones along this path. They are presented not necessarily chronologically, but in what would be considered a natural progression through ideas and understanding of the final conclusions. This may at times mean some results are directly obvious from others, but at the time of the experiment this was not the case.

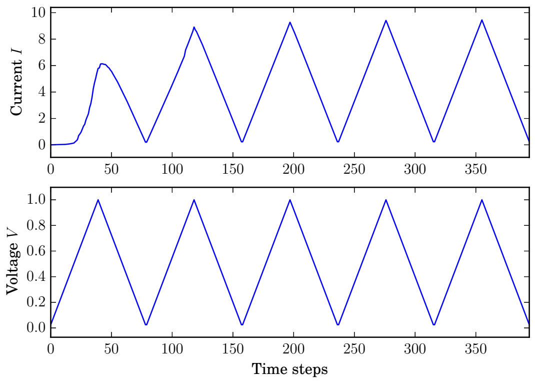
5.1 Replication

When designing software that models hardware, checking how the two compare is always a sensible first step. Because the atomic switch networks we are modelling are produced by the Nanotechnology Research Group at the University of Canterbury (NRG), we use the results they present in their paper and attempt to replicate exactly those results again [9]. Although they present many different variations over different tests on parameters in their paper, we focus on a few illustrative cases here.

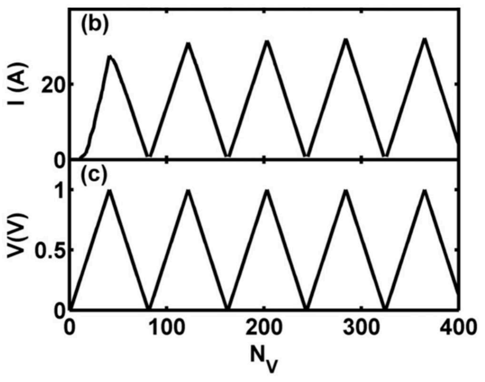
Fostner and Brown illustrate the response of the current through the atomic switch network by plotting the current over time as it responds to a series of voltage ramps from 0 V through to 1 V. Figure 5.1a is the same plot generated by our network simulation using parameters as for their Figure 2(b,c), repeated here as Figure 5.1b. Note the similarities in the key features—steady responses to the input voltage, delay on the initial ramp, and the stunted first peak. This is an exceptionally promising first start, but does not detail the conductance of the network of atomic switches, the most important characteristic of a neuromorphic system.

Fostner and Brown also present figures as in 5.2b showing how the conductance of the chip changes through a single voltage ramp from 0 V through to 0.5, 1, or 5 V. This is to test how the network behaves in the long term, once a "steady" or "saturated" state is reached. Figure 5.2a presents the same plot generated by our simulator for a chip with coverage 0.65. We see how the low switch-up probability noticeably lags behind the other two curves, whereas the 10 % and 80 % curves track closely, with 1 % lagging behind slightly. The similarities are sufficient that we can confidently conclude that the simulation is an accurate model of the work by the NRG.

A test that had incredibly successful results in the literature was by Pershin and Di Ventra, tasking the network with finding the shortest path through a maze [29]. This problem makes



(a) Results using our simulation.



(b) Figure 2(b,c) in [9].

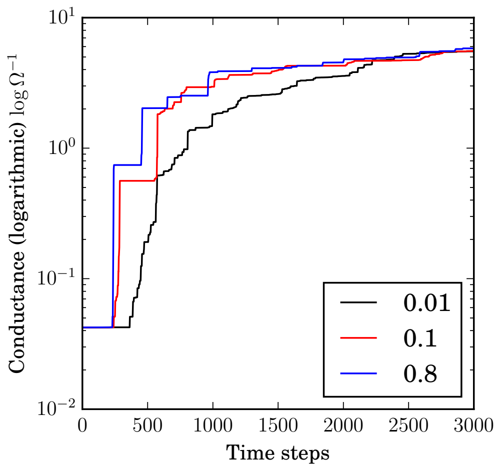
Figure 5.1: The current through the chip (coverage 0.65) as voltage ramps between 0 V and 1 V are applied. Actual current values are different due to different G_{max}.

intuitive sense, as memristors can scale their resistance based on the current that flows through them, so the shortest path will have the lowest resistance and so have the most current. To confirm that our network was capable of a task it should so clearly be able to manage, we gave it the same challenge. As expected, it completed this challenge with no difficulty. The downside of this approach is that the problem being solved is not general-purpose, but in fact encoded into the hardware of the chip. Thus although such a problem is trivially solvable with these networks, it in no way helps the goal of building a general-purpose learner. We do not agree with Pershin and Di Ventra that this technique can be used to efficiently solve the travelling salesman problem.

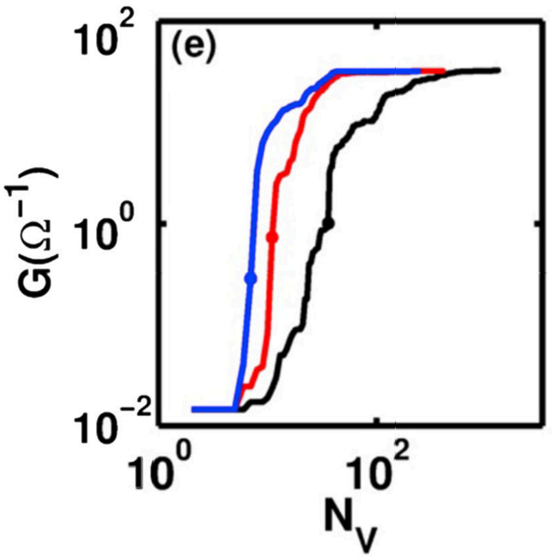
One avenue of research that has seen repeated interest is that of *associative learning*. Associative learning is when the current input for a given response is provided concurrently with a new input, and so the new input is associated with the response by the learner. The most famous example is Pavlov's Dog, in which a dog is trained to associate a bell with food. The same experiment can be conducted with artifical learners, and our neuromorphic reservoir learners. Two forms of associative learning with reservoir neuromorphic networks exist: the first where memristors are simply the weights between digital neurons, i.e. the reservoir is heterogeneous; and the second where the network consists solely of memristive hardware, i.e. the reservoir is homogeneous.

Pershin and Di Ventra describe a heterogeneous network of digital neurons connected by memristors acting as a neural network [28]. This network is shown to be capable of associative learning using food and sound signals to a salivation response. Importantly, this is achieved without explicitly associating the sound input to the salivation response. However this success does not transfer over to our large networks of memristive hardware because the networks we simulate are homogeneous, lacking the digital neurons. The digital neurons also *spike*, meaning that once a certain threshold is passed the neuron will send a short pulse of current both forward and backward. This makes the network entirely unlike our own, combined with the lack of random assembly meaning that the successes reported by Pershin and Di Ventra have little bearing on this research project.

Given how dissimilar the work of Pershin and Di Ventra is to our work, to find the work from Kulkarni and Teuscher so similar was surprising [20, 21]. This work describes a randomly assembled network of memristors, with no mention of digital neurons, only junctions. Kulkarni and Teuscher present a working associative learner using their simulation, some-



(a) Results from our simulation



(b) Figure 6(e) in [9].

Figure 5.2: The conductance of the chip (coverage 0.65) as a voltage ramp up to 1 V is applied. Comparable to Figure 6(e) in [9].

thing that we were unable to recreate. No combination of memristors or atomic switches we tested was capable of performing as required in this test. Even ESNs were incapable of learning in the way described. Because this paper is the only one found to suggest these results, we consider it anomalous.

Much of the research to date has been directed towards using heterogeneous networks of memristive components mixed in with digital neurons. This has the distinct disadvantage of no longer being randomly assembled, and so is more difficult to construct, is specific to the problem being solved, and also requires a hardware model of a neuron. Success in homogeneous networks would significantly reduce the difficulties associated with neuromorphic hardware production, but it does make significant changes to the assumptions of the network. Heterogeneous networks are able to indefinitely delay signals in neurons, add or remove voltage in these neurons, and in doing so change conditions in equations such as Kirchhoff's Laws, as the circuit is no longer a closed system. Because of the starting assumptions of this project, we do not simulate heterogeneous networks, and instead focus on the homogeneous atomic switch networks constructed by the NRG.

5.2 Results

One of the most fundamental machine learning tests is the MNIST database, a collection of 8×8 pixel greyscale images of handwritten digits. This dataset tests the time-independent learning ability of the reservoir, a test that it is not expected to perform exceptionally well on. This is because the network is designed to learn temporal datasets. Nevertheless, in an attempt to be thorough, we explore the time-independent learning potential of the reservoir, supposedly powered by updating weights in the reservoir itself.

The results of the 500-'neuron' reservoirs can be seen in Table 5.1, and similar tables for 100- and 200-neuron reservoirs is available in Appendix C. The ESN performed exactly as well as its readout layer, because the state-holding reservoir makes no difference when state is irrelevant. Thus it reports precision and recall in the region of 80 % to 90 %. We

Table 5.1: Precision (left) and recall (right) for 500-'neuron' learners with different digits, averaged over ten trials.

Digit	ESN		Memristors		Atomic Switches		Resistors	
0	0.9648	0.9705	0.8140	0.7750	0.9158	0.9466	0.9217	0.9886
1	0.8718	0.8385	0.7740	0.5736	0.7712	0.7626	0.8673	0.8615
2	0.8943	0.9035	0.7608	0.6116	0.9059	0.9000	0.9398	0.8558
3	0.8547	0.8176	0.6614	0.5967	0.8076	0.7736	0.8549	0.8099
4	0.9522	0.8543	0.8623	0.7065	0.9215	0.8826	0.9371	0.8761
5	0.8664	0.8857	0.8537	0.6000	0.8145	0.7923	0.8733	0.8736
6	0.9468	0.9440	0.8011	0.7440	0.9034	0.9418	0.9235	0.9593
7	0.9464	0.8798	0.7674	0.8225	0.8993	0.8674	0.9178	0.9135
8	0.7702	0.8273	0.7030	0.5830	0.7420	0.7227	0.8130	0.8273
9	0.7845	0.8837	0.7762	0.6359	0.7177	0.7804	0.7937	0.8500
Mean	0.8852	0.8805	0.7774	0.6649	0.8399	0.8370	0.8842	0.8816

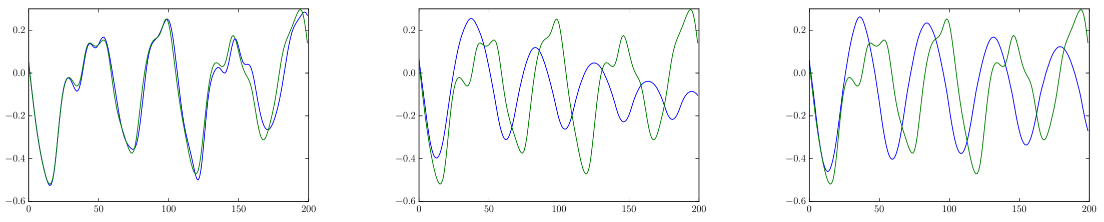
see comparable results from all the neuromorphic learners, including the resistor network, meaning that all the learning is occuring in the readout layer (which is identical for each learner). We can hence consider this figure to be correct, as linear regression can achieve up to about 90 % with MNIST. Note that the neuromorphic reservoirs are, on average, slightly lower-scoring, with the exception of the resistor network. This is likely because the underlying reservoirs are updating their conductances, and so the readout layer is not in fact trying to learn a single function, but a progression over time of linear functions, something it cannot do.

Others have attempted such tests with neuromorphic reservoirs. Querlioz et al. report accuracy of 81 % to 93 % [31], not markely different than the results presented here. However they compare their results to that of neural networks achieving accuracy in the mid-to-high 90's, considering them close, and therefore similar learners. We disagree with this statement, and conclude on the same data that these learners are powered primarily by their readout layer when working with time-independent datasets, and that the reservoir itself plays little to no role in the learning. Because the primary nature of the reservoir is to provide state[1], this conclusion is neither surprising nor concerning.

The Mackey-Glass test is a useful test of memory and temporal learning. We consider only the case when the memory length parameter τ is set to the smallest possible value 17, for the simple reason that none of the neuromorphic reservoirs were able to successfully learn in this 'easiest' case. We train the model with a sequence of inputs, and then start a feedback loop to let deviations in the past predictions accumulate. We can see in Figure 5.3a how a learner should be able to query its own memory/state to predict the future. Note how it starts tracking the true curve very closely, only to slowly drift further off over time.

The neuromorphic simulations were less successful, as shown in Figures 5.3b and 5.3c. Instead of slowly drifting away over time, the neuromorphic hardware essentially instantly forgets the complexities of the curve and instead settles in to a sine wave. This pattern is repeated by all the memristor and resistor learners, suggesting that the only reason there is a sine wave is because of the linear classifier. Thus there is no sufficiently rich state in the neuromorphic reservoir for the readout layer to tap in to, and thus there is not the memory

[1]This is not the only purpose of a reservoir, as the overall goal is for it to transform a non-linearly separable input into a feature space where it is linearly separable. However it does this by spreading the data out through time, which is not useful in a time-invariant problem.



(a) ESN **(b)** Memristors **(c)** Resistors

Figure 5.3: The predicted Mackey-Glass curves, blue (dark grey), plotted against the true curves, green (light grey), for each type of reservoir.

Table 5.2: The Mackey-Glass correlation distance for learners with reservoir sizes between 100 and 500 'neurons,' averages over ten trials.

Size	ESN	Memristors	Atomic Switches	Resistors
100	0.2261	0.8794	–	0.6333
200	0.0572	0.8319	–	0.6142
500	0.0509	0.7365	–	0.6849
Mean	0.1114	0.8159	–	0.6441

which is essential to solving the Mackey Glass problem. For any reservoir size, the atomic switch networks failed to learn, producing an unbounded model.

Testing more systematically, we can build Table 5.2, showing how the learners compare when tasked with the Mackey-Glass test. These results support what the plots have already told us. Remembering that a correlation distance d_{corr}, Equation (4.11), closer to 0 is better, we can see a clear disparity between the two types—ESN performing well with a correlation distance on average at 0.2280, while the neuromorphic simulations performed significantly more poorly, the correlation distance around 0.8. Such a distinct difference is immediately notable, and for reference the correlation distance between the Mackey-Glass curve and a horizontal line along zero is 1, so the neuromorphic learners are performing only marginally better to not predicting at all. Interestingly, the resistors yield a *better* score than we see from the memristors. We discuss this more in section 5.3, but as before this is because the resistors do not change and violate the assumptions of linear regression.

The results presented above are not what we had hoped. Ideally, this new network would be an exceptional learner, but this is not the case. Instead, we have a network that looks like it has all the features we need, but is incapable of learning. The obvious follow-up question is "why?" What features, or lack thereof, make the homogeneous neuromorphic networks perform so poorly at the "simple" Mackey-Glass test? What can be done about this?

5.3 Memory

A reservoir neural network is useful because it learns temporal datasets, whereas a feed-forward neural network would not. This is because the reservoir is able to maintain state, which is a source of memory. Here we outline the sources of memory, and discuss why it is they provide memory. Because implementations of reservoir neural networks using memristive hardware lack three of the four sources of memory described below, we constructed variations on the default ESN to explore what influence their loss might have. We outlined these learners in Figure 4.2, and will shortly present summary of the influence that these restrictions have. Because the atomic switches were incapable of producing a bounded model in the previous tests, this section will focus attention to memristor networks.

We have identified four sources of memory in an ESN: leaking, cycles, loops, and the discrete time steps. Leaking is the property of either (a) having the state from the previous time step leak forward in to the current time step, or (b) having the present state be forgotten partially making room for the past state. Cycles are when there is a sequence of neurons $n_1 n_2 \dots n_k n_1 n_2 \dots$. Loops are an edge that connects neuron n_i back to itself. The discrete time steps are when the state of a neuron at time t receives information from its neighbours from time $t-1$. This final property works in tandem with the first two to exploit the state of the network and provide the memory so vital in its power.

5.3.1 Leaking

Leaking is an inherently "external" property, and is not a reflection on the reservoir itself. Often denoted by the constant α, we mix the past input and the present input using the convex function

$$\mathbf{x}(t) = (1-\alpha)\mathbf{x}(t-1) + \alpha\tilde{\mathbf{x}}(t), \tag{5.1}$$

where $\tilde{\mathbf{x}}(t)$ is the pre-leaking state of the reservoir at time t. Hence α is a proportion, in this case representing the influence of the present input against the memory of past inputs.

Leaking provides memory because it creates a weighted exponential average of history in the readout layer. Thus the individual neurons in a reservoir follow the trend of the input, with the influence of short-term trends controlled by α. By spreading the input over the network, each region will experience different trends, and so we receive a rich variety of weighted averages.

Because leaking requires just the past output and the present output, it can be approximated in any readout layer. This makes it a valuable source of memory because, regardless of the reservoir type, we can guarantee its presence. Readout layers are typically implemented in software due to needing training, so leaking is stored as two variables: α, and $\mathbf{x}(t-1)$.

5.3.2 Cycles

Cycles provide a reservoir with 'infinite' memory. By having the input from past times becoming available again essentially for free, the reservoir can continue to mix in this information with no concern for when it was introduced. That is, a cycle of length k provides access to the output of the same neuron from time $t - dk$, with $d \in \mathbb{N}$. This long-term memory supports the echo property, Equation (2.8), that is so important to the reservoir.

Removing cycles is an important research question, because hardware implementations are unable to recreate cycles. Kirchhoff's Voltage Law, Definition 3.2, limits the amount of energy in a circuit, and forces conservation. That is, a junction is unable to amplify a signal, and so there cannot be cycles in the network. Having cycles would imply an infinite sequence of groups where the potential difference drops forever, leading to an impossible infinitely-descending structure:

$$V_1 > V_2 > \cdots > V_k > V_1 > \cdots \implies V_1 > V_1 \ \lightning \tag{5.2}$$

If cycles were to form, energy would cycle forever and become infinite, something not possible in a physical circuit.

Now, having removed cycles, infinite mixing of inputs is not available to every neuron, but infinite mixing of input at neuron n for input to neurons $m < n$ is available because we have not yet excluded loops. By modifying input to be repeated (i.e. $\mathbf{u}(t) \mapsto [\mathbf{u}(t); \mathbf{u}(t)]$), the inputs to neurons $m < n$ are also available at neurons $o > n$. Thus having loops can be made equivalent to having cycles, although particular mixes may not be available within the same number of time steps. This is important as a device which mixes the previous voltage across a memristor with the present voltage is conceivable.

As an illustrative example, consider a network that once contained the cycle of two neurons m and n such that $m \to n \to m$. Normally we could infinitely mix $u_m(t)$ with $u_n(t-2k-1)$ and $u_m(t-2k)$ for any natural number k, and vice versa, by allowing the inputs to cycle around each other. By removing cycles, such a structure is unavailable. Instead, we can simulate it with $m \to n \to m' \to n'$ such that $u_m(t) = u_{m'}(t)$ and $u_n(t) = u_{n'}(t)$, and every neuron also loops back into itself. It is now possible to mix $u_m(t)$ with $u_n(t-k)$ and $u_m(t-k-1)$ for any natural number k, as it now occurs further back in the network, and the cycle acts as an infinite internal delay mechanism for the input. This is a stronger guarantee than necessary, but does ensure the desired effect of cycles.

As mentioned, Kirchoff's Voltage Law implies conservation of energy, but the restriction of conservation of energy is not a restriction at all. It limits a reservoir in the same manner as the spectral radius, the spectral radius being the largest absolute eigenvalue of the weights matrix. By ensuring that a neuron's outputs sum to one, we have effectively forced each column in the weights matrix to sum to one. The eigenvalues of matrix $\mathbf{W}$ are the same for $\mathbf{W}^\top$, so we can consider the matrix $\mathbf{W}^\top$ with row sums equal to 1. For some $\mathbf{v}$, we have

$$\mathbf{W}^\top \mathbf{v} = (\mathbf{w}_1^\top \mathbf{v}, \mathbf{w}_2^\top \mathbf{v}, \ldots)^\top = (\mathbf{w}_1 \cdot \mathbf{v}, \mathbf{w}_2 \cdot \mathbf{v}, \ldots)^\top \tag{5.3}$$

Given that $\mathbf{w}_i \cdot \mathbf{v} = \|\mathbf{w}_i\|_2 \|\mathbf{v}\|_2 \cos\theta$, $\|\mathbf{w}_i\|_2 \leq 1$, and $-1 \leq \cos\theta \leq 1$, the largest absolute scaling possible by $\mathbf{W}^\top$ (and thus also by $\mathbf{W}$) is 1. Hence conservation of energy is equivalent to specifying a spectral radius of at most one. This is not an issue: ESNs are only guaranteed to work for spectral radii below one [16].

5.3.3 Loops

Loops are a source of memory for the reservoir, again contributing to the infinite memory and echo property. Because the neuron now has explicit access to its own output at time $t-1$, it creates a type of weighted average, effectively giving each neuron total memory of past inputs. Cycles give the same effect, but the tighter effect of the loop is more easily emulated in hardware solutions by sensors and external voltage sources.

As shown by Čerňanský and Makula, removing both cycles and loops reduces an ESN to a feed-forward network with delayed-time inputs [6]. The memory of the network is limited by the longest chain. The network was still capable of solving the typical sorts of problems such as Mackey-Glass because the memory requirement is by convention set at 17 steps, and the reservoirs are trivially made larger than this. Removing just one of loops or cycles will not cause the same reduction in expressive power for temporal datasets. By removing only loops and not cycles, there is no immediate loss of power—any memory a loop supported is replicated with a cycle, but with a k-step delay, where k is the length of the shortest cycle through a neuron. Thus learning may slow, but not stop.

Consider a simple network of two neurons connected by a directed edge in both directions. If no loops are available, it is not immediately possible to mix the input to neuron i at time t, denoted $u_i(t)$, with $u_i(t-1)$. But we can mix $u_i(t)$ with $u_i(t-2)$. Thus the length of the cycle through neuron i is two, so there is a two-step delay in the network. In the meantime, neuron j is mixing $u_i(t-1), u_i(t-3), \ldots$. The readout layer can mix both streams, thus mixing $u_i(t)$ for all t. This scales appropriately for cycles of length k.

Algorithm 1 Propagate the input $\mathbf{u}(t)$ over the reservoir defined by $\mathbf{W}$

```
1:  procedure PROPAGATE($\mathbf{W}$, $\mathbf{W}^{\text{in}}$, $\mathbf{u}(t)$)
2:      $\mathbf{v} \leftarrow \mathbf{W}^{\text{in}}\mathbf{u}(t)$
3:      $\mathbf{o} \leftarrow (0, 0, \ldots, 0)^\top$
4:      for all $n \in$ toposort($\mathbf{W}$) do
5:          $s \leftarrow v_n$
6:          for all $m \in$ predecessors($n$) do            ▷ finds all nodes with edges into $n$
7:              $s \leftarrow s + o_m \mathbf{W}_{n,m}$             ▷ $\mathbf{W}_{n,m}$ is the weight from $m$ to $n$
8:          end for
9:          $o_n \leftarrow \tanh(s)$
10:     end for
11:     return $\mathbf{o}$
12: end procedure
```

5.3.4 Discrete time steps

The discrete time nature of an ESN is the fundamental feature of its memory. This is also a difficult feature to replicate in hardware. Because a circuit will have the electricity pass through at significant fractions of the speed of light, no matter how rapidly we switch the input voltage, we are essentially saturating the network with the same signal millions of times before switching.

Because of this speed disparity, a hardware network will essentially not contain discrete time steps, instead it will function more like a traditional feed-forward neural network, which we will call the one-hop reservoir, where the input $\mathbf{u}(t)$ is influencing the entire network at time t, but inputs $\mathbf{u}(s)$ from times $s < t$ are not in the network. The difference is now, there is no new information written to the network before the propagation is complete. Because of this distinct termination, the network is not allowed to have cycles or loops. We can propagate the information using Algorithm 1.

But why does the lack of discrete time matter? Because of how a reservoir is defined, it must have a clear boundary of past and present, and by having the network become saturated at each time there is effectively no past. The past outputs are the defining feature of recurrent neural networks, because the ability to use past knowledge is what enables the reservoir to maintain state, which in turn provides the ability to learn temporal functions.

Because there is now no state in the network beyond the leaking rate, the network will be unable to learn any function requiring knowledge of previous time steps. Essentially, we remove the echo property. The state now depends solely on the random initial weights, not the history of previous inputs as required by an ESN. This network is now an untrained feed-forward neural network. Hence this reservoir is incapable of learning any of the time-series problems it was designed to solve. While there are potential applications for traditional machine learning, by training an equivalent neural network in software and 'burning in' the weights to hardware, this is not a suitable use for memristors—they will update their weight, and move away from their desired weight.

This comparison is not fair, because a network of memristors *does* maintain a state, because the weights *do* get updated. The question then becomes does the memristor's state act as a suitable substitute for the ESN's discrete time steps? The answer would seem to be no. By making the ESN have a 'wobbling' weights matrix to simulate the updating conductances of the memristors and switches, we handicap the readout layer by removing the underlying assumption of regression—for a given input x, there is a function $f(x)$ that

Table 5.3: Significance levels between distributions of correlation distances for different learners. Stars signify 95 %, 99 % and 99.9 % confidence intervals.

	Discrete vs One-hop		Discrete vs Memristor		One-hop vs Memristor	
Size	p	Sig.	p	Sig.	p	Sig.
50	0.0160	*	0.0001	***	0.3381	
100	0.1668		0.0054	**	0.2224	
200	0.0190	*	< 0.0001	***	0.0593	
500	0.0440	*	< 0.0001	***	0.0185	*
750	0.0291	*	0.0001	***	0.1068	
1000	0.0022	**	< 0.0001	***	0.4198	
1500	0.0036	**	0.0003	***	0.7328	
2000	0.0237	*	0.0006	***	0.3445	

we attempt to find. Because f is a function, each x uniquely maps to some y. By changing the weights matrix, we change the function we are trying to fit, and so prevent the linear regression from successfully fitting the training data.

By generating a large selection of memristor networks and feed-forward conservative ESNs that are both discrete-time and one-hop with a range of reservoir sizes, we can explore whether they all exhibit similar learning tendencies, and if not which networks behave most similarly. We generated ten reservoirs of each learner, with reservoir sizes ranging from 50 to 2000 neurons, trained the reservoir to predict the Mackey-Glass $\tau = 17$ problem set, and calculated the correlation distance between the output curve and the expected curve for the next 200 steps, calculated using Equation 4.11. Before running statistical tests, we throw out the "failed" learnings. We determine these by calculating the area between the output and expected curves. We consider any area that exceeds 10^{10} as a failed learning.

A one-way ANOVA test, where the grouping is the pair (learner, size), reveals there is a significant difference between the groups ($F_{24} = 6.62$, $p < 0.0001$). To see where these differences actually occur, we perform a Student's t-test between two learners at each size, and the result of this is in Table 5.3. Because there are a large number of t-tests conducted, and a Bonferroni correction is too conservative, the chance of making a Type-I error rises. Hence we consider more the "broad strokes" rather than the precise p-values. The first thing that is clear is that distinguishing between the one-hop ESN and the memristor is difficult, with only one significant result out of all the reservoir sizes. This is in contrast to what occurs between the discrete-step ESN and both other learners, in particular memristors. We can distinguish the discrete-step ESN from either of the other learners with good consistency.

These differences become clear when we plot the predicted curve alongside the actual curves they were intended to match. The examples in Figure 5.4 show how each reservoir fares as a predictor, and makes clear why the memristors and one-hop ESNs are so difficult to tell apart. The correlation distances for each prediction are: 0.13638 for the discrete ESN; 1.30299 for the one-hop ESN; and 0.62263 for the memristor reservoir. The closer the two curves follow one another, the smaller the correlation distance between them.

5.4 Other approaches

The above results paint an unfortunate picture for homogeneous neuromorphic reservoirs attempting to serve as machine learning systems. Because of this, we now turn to ways around this problem. Several paths forward exist, and below we evaluate the strengths and

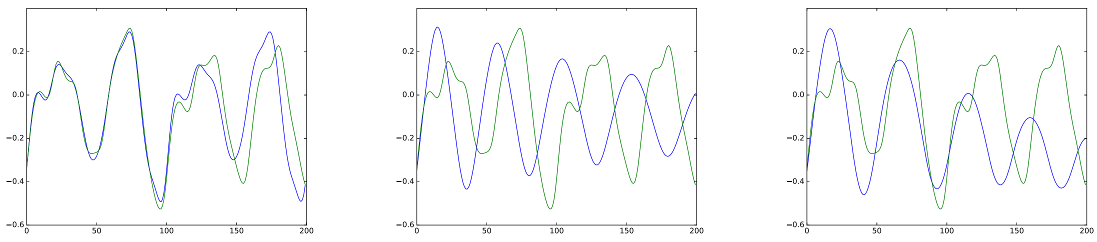

(a) Discrete ESN **(b)** Onehop ESN **(c)** Memristor Reservoir

Figure 5.4: The predicted Mackey-Glass curves, blue (dark grey), plotted against the true curves, green (light grey), for each type of reservoir.

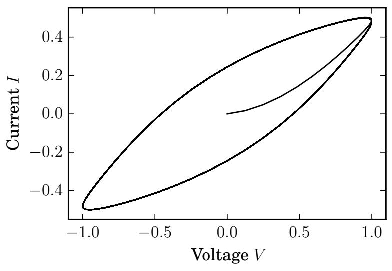


Figure 5.5: The "I-V" hysteresis from an ESN neuron. Note the lack of pinch through the origin.

weaknesses of each. These range from systems we know perform well, through to ideas that have not yet been attempted, but offer hope that this hardware will still be useful.

An iconic part of memristive hardware such as memristors and atomic switches is the pinched hysteresis, which we presented back in Figure 2.4. This plot is generated by applying voltage as a sine wave and measuring the response current. If we consider a similar plot for ESNs relating input and output together, we can generate the curve in Figure 5.5. An important distinction between this curve and the hysteresis seen in memristive components is the lack of a pinch through $(0, 0)$. The physical interpretation for this is interesting, in that we can read it as when there is no input voltage there is still current flowing through the network. This is quite different to any of the components we have looked at so far, but is not an unreasonable phenomenon. In fact, such a curve is exactly the same kind of hysteresis we would observe from a capacitor or inductor. Using these components instead of the memristive components we have here may produce different results because both are able to act as a delay mechanism.

Alternatively, instead of trying to fix the discrete time problem, we can consider whether it is even a concern. Because of the drawbacks of homogeneous networks with discrete time, we can instead use them to solve problems with no time dependence. One of the largest classes of such a problem is graph search, having applications in almost every problem we have attempted to solve. The behaviour of memristors and atomic switches suggests that these networks could perform parallel path search significantly faster than we can currently achieve in software.

As mentioned at the beginning of this chapter, we are able to recreate the maze-solving work by Pershin and Di Ventra [29]. However, this approach does have a significant drawback—structure. The layout of the hardware is dictated by the problem. Although there are ways around this, such as the ability to turn off certain paths in software when describing the problem, the underlying issue is that the board needs to be a regular—or at least predictable—grid. This is not the case with the hardware from the NRG, as one major benefit of their approach is the random self-assembly.

Because most of the issues with the neuromorphic networks identified above are tied to the fact that they are homogeneous, the next obvious step would be to work with heterogeneous networks instead. Rather than having just memristors or atomic switches in the reservoir, include digital neurons as well. These could be of varying complexity, from simple perceptron-style neurons through to leaky integrate-and-fire models, the latter of which are used in the standard liquid state machine formulation by Maass et al. [25].

Again, this change is not without its downsides. If digital neurons are introduced into the network, we must question what value the memristors and atomic switches have in learning—if adding neurons makes all the difference, why not have just neurons? Additionally, random assembly again becomes difficult. If we need to place these digital neurons in the network, can we still rely on stochastic depositions to generate suitable reservoirs? Such questions make it difficult to assess the future of memristors and atomic switches in heterogeneous reservoirs.

All of these ideas continue to link back to the concept of reservoir computing. But by changing this assumption we can change the designs that we can have. There are two major non-reservoir approaches that we can take: pre-training single-purpose neural networks, or neural networks with variable resistors as weights. Again, all the following discussion works against random assembly, but such discussions must be had to provide a thorough overview of avenues forward from here.

Two key reasons for moving neural networks to hardware are speed and energy efficiency. Neither of these demand that the neural network be better than current designs, nor that it be a general purpose learner in the sense that it could learn anything. Often when we create neural networks, they are feed-forward, and trained for a single purpose and then left as is until the requirements change. Thus we have to ask, if we only plan on training it once, why does this have to be done in hardware? By first building the neural network in software and translating that same network in to hardware we potentially get all the benefits of hardware neural networks with none of the difficulties of self-updating hardware. Having specifically designed hardware would be beneficial for large scale work where the same neural network is used thousands if not millions of times.

If a neural network does need to be updated frequently, then this train-once approach is clearly not suitable. Instead, rather than setting the resistors' resistance at manufacture, construct the network from software-controlled variable resistors. Such a network could then be trained in a combination of hardware and software, and then run entirely in hardware, bringing the benefits of both software and hardware together.

If we consider the memory component of reservoir neural networks important, we can instead consider moving to a model hinted to by Čerňanský and Makula, where there is an explicit delay mechanism in front of the reservoir storing the state, rather than the network itself [6]. Such a delay mechanism would trivially be controlled in software, and although this would remove the infinite memory so appealing in ESNs, a sufficiently large delay mechanism would render this point moot.

6 Conclusion

"The end of all our exploring will be to arrive where we started and know the place for the first time."
— T. S. Eliot

This project spans physics, statistics, mathematics, and computer science, drawing on work from each of these fields. We present contributions of value both to physics and engineering by helping to guide future research and development of hardware, as well as computer science and in particular artificial intelligence, by identifying the key components of reservoir computing that were otherwise not apparent.

6.1 Summary

A fundamental part of this project was to produce a sufficiently accurate simulator of the atomic switch network hardware produced by the Nanotechnology Research Group at the University of Canterbury. This work was helped by existing code from Fostner and Brown, but significant rewriting occurred in attempts to improve the efficiency, speed, and maintainability of the codebase. Although this does mean moving away from Matlab, the resulting blend of Python, SciPy, and Fortran—languages already established in the science community—is equally approachable and should serve as a solid foundation for future work.

The most significant contribution in the simulations was the development of the statistical generation method. Rather than spending time constructing the boards out of individually simulated particles, we present a method to generate the board in a fraction of the time by drawing the board parameters from probability distributions modelled on existing data. The result is a board that is appropriate for simulations without the time-intensive deposition process, enabling rapid iteration and larger board sizes.

Because of the design of the simulations, we allow arbitrary tunnels in our network, meaning that the simulations are not restricted to the hardware components we had in mind when implementing them. Thus further work exploring ideas such as capacitive networks or inductor networks is viable, and potentially time-inexpensive due to the limited amount of new code necessary.

Once the simulations were working, the challenge became producing a reservoir neural network using them as the reservoir. We present a modular framework for building reservoir learners, where each component can be swapped in or out quickly, enabling a rapid-prototyping approach to development. This system enabled us to produce learners in a wide variety of configurations, so we could see what works, and what doesn't. This is important because such large homogeneous neuromorphic reservoirs are rare in the research, so "ideal" parameters are difficult to find.

Due to the failure of the neuromorphic reservoirs to successfully learn, we were driven to find the underlying cause. By systematically restricting the reservoir in an ESN, we were able to identify four key sources of learning: leaking, loops, cycles, and discrete time. While leaking is a simple addition to any network, the other three are problematic in homogeneous networks of fundamental circuit components. By considering individually and in combination what removing these features would do to the reservoir, we can produce a picture of how a reservoir operates and the underlying assumptions and requirements.

Consider first the loops and cycles present in a reservoir. When both are removed as in a feed-forward ESN, there is a notable change in the network in the form of losing "infinite memory." But by removing only a single feature of loops and cycles, the network is able to maintain infinite memory, and with only minor alterations we are able to simulate either loops or cycles using the other. This result means that adding both features is not necessary, reducing the potential complexity for hardware manufacturers.

Finally we identified the most important feature of a reservoir—discrete time. Without discrete time, the reservoir is essentially flooded with eternal history of a single state, and any knowledge of the previous state is "drowned out" by the new information. This result leads us to believe that a reservoir made solely of fundamental circuit components is limited in its applicability as a reservoir, and instead efforts should be directed towards alternative approaches such as networks with digital neurons or explicit hardware delay mechanisms.

6.2 Limitations

Although we have strived to be thorough, any project of this size will inherently be limited by both time and scope of questioning. As such there are questions that we have been unable to address fully, or have been unable to pursue as deeply as we would wish.

A concern that has risen more than once during this project was the suitability of Kirchhoff's laws. Like any physical law, Kirchhoff's laws come with their own set of assumptions that must be adhered to in order for the results to still be meaningful. One if these is the "lumped element" assumption, which is where each component is assumed to be uniform, and the timescales at which the electromagnetic waves propagate is significantly smaller than the timescales of interest. As became clear when the discrete-time memory factor was discovered, we may well need to work on the timescales of electromagnetic propagation. Present hardware is unable to switch at the speeds required (in the order of picoseconds), but even if it were the simulations developed here are entirely unsuitable.

Two other concerns present themselves in the statistics presented. The first is small sample sizes. Although we have endeavoured to make the simulations as efficient and fast as possible, we still have to perform a large number of calculations while running up against an unfortunately large complexity. This means that the sample sizes are smaller than we would like, although because the results are so clear-cut we do not feel that larger sample sizes would have any impact on the findings, only on the confidence in these results.

Similarly, the number of parameters we have tested is less than ideal. Although we have presented a number reservoir sizes over a number of tests, more work is suggested in finding the ideal balance of parameters such as the leaking rate and the normalisation constant. With more time these concerns can be addressed, and again they are unlikely to change the underlying result, only improve our confidence in the results presented.

6.3 Future work

Like any research, we have generated as many questions as we have answered. Some of these are of immediate interest and a direct development of the work here, while others are more long-term goals of interest to the field in general. We have discussed some of these ideas in more detail in Chapter 5.

Although the work here has focused on memristors and atomic switches (and resistors), there are other potentially suitable components. When considering the plot of an ESN neuron in Figure 5.5, we see hysteresis not unlike that of a capacitor or inductor. Using such components for neuromorphic computing would be an interesting avenue for future research, as capacitors in particular may be able to act as a delay mechanism in the network and overcome some of the time difficulties.

Alternatively, rather than trying to add state to the reservoir, we can explicitly add state to the input. Although this is not ideal, we can certainly still reap the benefits of a hardware neural network even if we do not gain the power of a reservoir neural network. Such explicit delay mechanisms will also mean that traditional feed-forward neural network training algorithms such as back-propagation are suitable, meaning that training is simplified.

Of course, there is no reason that we are restricted to homogeneous networks in the first place. By allowing networks of heterogeneous components, notably some type of digital neuron, we are able to overcome many of the issues outline in this report. Because the neurons will be able to act as sources and sinks of energy, or even delay the propagation of energy through the network, the restrictions on cycles, loops, and discrete time are all removed. Thus by adding digital neurons to the network we may be able to reach the power we want in neuromorphic reservoirs. Note that the simulation here is incapable of generating these kinds of networks.

Another area of exploration that we did not have much time to investigate was alternative information encodings. This project uses the naïve encoding from one input value to one voltage level, but this is not the only possible encoding. An alternative representation we considered was using sine waves and varying the amplitude as the input level, or encoding the input as frequency. Both of these input encodings require an alternating current simulation, which we do not have. The question then becomes how this would be stored as state for the network, leaving many avenues for future research.

In this project, we have demonstrated how we can reduce an ESN to have the same predictive power as a memristor reservoir. The question remains as to whether the same approach can be applied in the other direction, to try and add features to a memristor reservoir so that it has the same learning power as an ESN. Early tests show that this is a nontrivial exercise due to the breakdown of Kirchhoff's laws when attempting to enforce discrete time. Further research would be an interesting and informative challenge to explore how this could be done.

Bibliography

[1] F. Akopyan, J. Sawada, A. Cassidy, R. Alvarez-Icaza, J. Arthur, P. Merolla, N. Imam, Y. Nakamura, P. Datta, G. J. Nam, B. Taba, M. Beakes, B. Brezzo, J. B. Kuang, R. Manohar, W. P. Risk, B. Jackson, and D. S. Modha, "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*, vol. 34, no. 10, pp. 1537–1557, October 2015.

[2] A. V. Avizienis, H. O. Sillin, C. Martin-Olmos, H. H. Shieh, M. Aono, A. Z. Stieg, and J. K. Gimzewski, "Neuromorphic atomic switch networks," *PLoS ONE*, vol. 7, no. 8, pp. 1–8, August 2012.

[3] C. M. Bishop, *Pattern Recognition and Machine Learning.* Springer, 2006.

[4] J. Bürger and C. Teuscher, "Variation-tolerant computing with memristive reservoirs," in *IEEE/ACM International Symposium on Nanoscale Architectures*, July 2013, pp. 1–6.

[5] J. Bürger, A. Goudarzi, D. Stefanovic, and C. Teuscher, "Hierarchical composition of memristive networks for real-time computing," in *Proceedings of the 2015 IEEE/ACM International Symposium on Nanoscale Architectures (NANOARCH15).* IEEE, July 2015, pp. 33–38.

[6] M. Čerňanský and M. Makula, "Feed-forward echo state networks," in *Proceedings. 2005 IEEE International Joint Conference on Neural Networks, 2005.*, vol. 3, July 2005, pp. 1479–1482.

[7] M. Čerňanský and P. Tiňo, *Artificial Neural Networks – ICANN 2007: 17th International Conference, Porto, Portugal, September 9-13, 2007, Proceedings, Part I.* Berlin, Heidelberg: Springer Berlin Heidelberg, September 2007, vol. 4668, ch. Comparison of Echo State Networks with Simple Recurrent Networks and Variable-Length Markov Models on Symbolic Sequences, pp. 618–627.

[8] L. O. Chua, "Memristor – the missing circuit element," *IEEE Transactions on Circuit Theory*, vol. 18, no. 5, pp. 507–519, September 1971.

[9] S. Fostner and S. A. Brown, "Neuromorphic behavior in percolating nanoparticle films," *Phys. Rev. E*, vol. 92, no. 5, p. 052134, November 2015.

[10] S. B. Furber, F. Galluppi, S. Temple, and L. A. Plana, "The spinnaker project," *Proceedings of the IEEE*, vol. 102, no. 5, pp. 652–665, May 2014.

[11] M. Hu, H. Li, Y. Chen, Q. Wu, G. S. Rose, and R. W. Linderman, "Memristor crossbar-based neuromorphic computing system: A case study," *IEEE Transactions on Neural Networks and Learning Systems*, vol. 25, no. 10, pp. 1864–1878, October 2014.

[12] M. Hutter, *Univeral Artificial Intelligence.* Springer, 2005.

[13] G. Indiveri and S.-C. Liu, "Memory and information processing in neuromorphic systems," *Proceedings of the IEEE*, vol. 103, no. 8, pp. 1379–1397, August 2015.

[14] G. Indiveri, B. Linares-Barranco, T. J. Hamilton, A. van Schaik, R. Etienne-Cummings, T. Delbruck, S.-C. Liu, P. Dudek, P. Häfliger, S. Renaud, J. Schemmel, G. Cauwenberghs, J. Arthur, K. Hynna, F. Folowosele, S. SAÏGHI, T. Serrano-Gotarredona, J. Wijekoon, Y. Wang, and K. Boahen, "Neuromorphic silicon neuron circuits," *Frontiers in Neuroscience*, vol. 5, no. 73, pp. 1–23, May 2011.

[15] G. Indiveri, B. Linares-Barranco, R. Legenstein, G. Deligeorgis, and T. Prodromakis, "Integration of nanoscale memristor synapses in neuromorphic computing architectures," *Nanotechnology*, vol. 24, no. 38, p. 384010, September 2013.

[16] H. Jaeger, "The "echo state" approach to analysing and training recurrent neural networks," German National Research Institute for Computer Science, GMD Report 148, January 2001.

[17] S. H. Jo, T. Chang, I. Ebong, B. B. Bhadviya, P. Mazumder, and W. Lu, "Nanoscale memristor device as synapse in neuromorphic systems," *Nano Letters*, vol. 10, no. 4, pp. 1297–1301, March 2010.

[18] Z. Konkoli and G. Wendin, "A generic simulator for large networks of memristive elements," *Nanotechnology*, vol. 24, no. 38, p. 384007, September 2013.

[19] Z. Konkoli and G. Wendin, "On information processing with networks of nano-scale switching elements," *International Journal of Unconventional Computing*, vol. 10, no. 5/6, pp. 405–428, November 2014.

[20] M. S. Kulkarni, "Memristor-based reservoir computing," Master's thesis, Portland State University, 2012.

[21] M. S. Kulkarni and C. Teuscher, "Memristor-based reservoir computing," in *2012 IEEE/ACM International Symposium on Nanoscale Architectures (NANOARCH)*, July 2012, pp. 226–232.

[22] R. Legenstein and W. Maass, "Edge of chaos and prediction of computational performance for neural circuit models," *Neural Networks*, vol. 20, no. 3, pp. 323–334, April 2007.

[23] B. Linares-Barranco, T. Serrano-Gotarredona, L. A. Camuñas-Mesa, J. A. Perez-Carrasco, C. Zamarreño-Ramos, and T. Masquelier, "On spike-timing-dependent-plasticity, memristive devices, and building a self-learning visual cortex," *Frontiers in Neuroscience*, vol. 5, no. 26, pp. 1–22, March 2011.

[24] M. Lukoševičius, *Neural Networks: Tricks of the Trade: Second Edition.* Springer, 2012, vol. 7700, ch. A Practical Guide to Applying Echo State Networks, pp. 659–686.

[25] W. Maass, T. Natschläger, and H. Markram, "Real-time computing without stable states: A new framework for neural computation based on perturbations." *Neural Computation*, vol. 14, no. 11, pp. 2531 – 2560, November 2002.

[26] C. Mead, *Analog VLSI and Neural Systems*, ser. Addison-Wesley VLSI systems series. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., January 1989.

[27] D. Monroe, "Neuromorphic computing gets ready for the (really) big time," *Commun. ACM*, vol. 57, no. 6, pp. 13–15, June 2014.

[28] Y. V. Pershin and M. Di Ventra, "Experimental demonstration of associative memory with memristive neural networks," *Neural Networks*, vol. 23, no. 7, pp. 881–886, September 2010.

[29] Y. V. Pershin and M. Di Ventra, "Solving mazes with memristors: A massively parallel approach," *Phys. Rev. E*, vol. 4, no. 84, p. 046704, March 2011.

[30] D. Querlioz, P. Dollfus, O. Bichler, and C. Gamrat, "Learning with memristive devices: How should we model their behavior?" in *2011 IEEE/ACM International Symposium on Nanoscale Architectures*, June 2011, pp. 150–156.

[31] D. Querlioz, O. Bichler, P. Dollfus, and C. Gamrat, "Immunity to device variations in a spiking neural network with memristive nanodevices," *IEEE Transactions on Nanotechnology*, vol. 12, no. 3, pp. 288–295, May 2013.

[32] S. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach*, 3rd ed. Pearson, 2010.

[33] S. Saïghi, C. G. Mayr, T. Serrano-Gotarredona, H. Schmidt, G. Lecerf, J. Tomas, J. Grollier, S. Boyn, A. Vincent, D. Querlioz, S. La Barbera, F. Alibart, D. Vuillaume, O. Bichler, C. Gamrat, and B. Linares-Barranco, "Plasticity in memristive devices for spiking neural networks," *Frontiers in Neuroscience*, vol. 9, no. 51, pp. 1–16, March 2015.

[34] A. Sattar, S. Fostner, and S. A. Brown, "Quantized conductance and switching in percolating nanoparticle films," *Physical Review Letters*, vol. 111, no. 13, p. 136808, June 2013.

[35] R. F. Service, "The brain chip," *Science*, vol. 345, no. 6197, pp. 614–616, August 2014.

[36] R. A. Serway, J. W. Jewett, K. Wilson, and A. Wilson, *Physics*, Asia-Pacific ed., M. Veroni, Ed. Cengage Learning, 2013, vol. 2.

[37] H. O. Sillin, R. Aguilera, H.-H. Shieh, A. V. Avizienis, M. Aono, A. Z. Stieg, and J. K. Gimzewski, "A theoretical and experimental study of neuromorphic atomic switch networks for reservoir computing," *Nanotechnology*, vol. 24, no. 38, p. 384004, September 2013.

[38] A. Smith, "Simulating percolating superconductors," Master's thesis, University of Canterbury, 2014.

[39] J. E. Steif, "A mini course on percolation theory," 2009.

[40] A. Z. Stieg, A. V. Avizienis, H. O. Sillin, C. Martin-Olmos, M. Aono, and J. K. Gimzewski, "Emergent criticality in complex turing b-type atomic switch networks," *Advanced Materials*, vol. 24, no. 2, pp. 286–293, January 2012.

[41] D. B. Strukov, G. S. Snider, D. R. Stewart, and R. S. Williams, "The missing memristor found," *Nature*, vol. 453, no. 7191, pp. 80–83, May 2008.

[42] W. S. Zhao, G. Agnus, V. Derycke, A. Filoramo, J.-P. Bourgoin, and C. Gamrat, "Nanotube devices based crossbar architecture: toward neuromorphic computing," *Nanotechnology*, vol. 21, no. 17, p. 175202, April 2010.

Appendices

A Full Simulator Calculations

First we present the two important methods from the Python `MemChip` class. References to `fastchip` are calls to Fortran, the source for which is given afterwards.

Python

```python
def write(self, input_matrix: Iterable) -> Iterable:
    """ Encode the input_matrix as voltages for the neural
        network to process, and feed it to the network.
        Each row is a new input vector.
    """
    rows, _ = input_matrix.shape
    conductance_result = scipy.zeros(rows)
    current_result = scipy.zeros((rows, self.output_count))
    sensors_result = scipy.zeros((rows, (MemChip.sensor_grid_width *
                                         MemChip.sensor_grid_height)))
    for t, v in enumerate(input_matrix):
        current_matrix, _, conductance = self._feed_through_network(v)
        # print(current_matrix)
        if self._sensor_grid_enabled:
            sensor_result = fastchip.read_sensor_grid(
                self.sensor_grid,
                current_matrix,
                MemChip.sensor_grid_width * MemChip.sensor_grid_height)
        else:
            sensor_result = 0
        currents = scipy.sum(
            current_matrix[:, scipy.array([-(i+1) for i in
                                           range(self.output_count)])],
            axis=0)
        currents = scipy.fliplr(currents)
        currents = scipy.ravel(currents.sum(axis=0))
        conductance_result[t] = conductance
        current_result[t] = currents
        sensors_result[t] = sensor_result
        gc.collect()
    return (conductance_result, current_result, sensors_result)

def _feed_through_network(self, voltages: Iterable) -> Tuple[scipy.matrix,
                                                             scipy.matrix,
                                                             float]:
    """ Given an input, run it through the network
        and calculate the state of the network because of it.
    """
    for _ in range(self.tunnel.cycles):
        conductance_matrix = self.structure.read()
        current_shell = self.current_matrix.copy()
        # Create the G Matrix
        g_matrix = fastchip.g_matrix(conductance_matrix,
                                     current_shell)
        # Fill in the voltage vector
        v_vec = self.voltage_vector.copy()
        v_vec[v_vec > 0] = voltages
        # Solve the simultaneous equations from [38].
        try:
            out_vec = scipy.linalg.solve(g_matrix,
                                         v_vec)
        except Exception as e:
            # Does not stop exception, but shows useful information
            print(v_vec[self.number_of_groups:
                        self.number_of_groups+self.input_count])
            print(g_matrix)
            if not scipy.isfinite(g_matrix).all():
                # Show both infinite and NaN values
                oddities = scipy.logical_not(scipy.isfinite(g_matrix))
                print(g_matrix[oddities])
            print(self.structure.sizes)
            raise e
        voltage_matrix = scipy.matrix(fastchip.voltage_matrix(
            out_vec[:self.number_of_groups],
            self.structure.sizes == scipy.inf))
        current_matrix = scipy.matrix(fastchip.current_matrix(
            voltage_matrix,
            conductance_matrix,
            self.structure.sizes == scipy.inf))
        self.structure.apply(voltage_matrix, current_matrix)
    n = self.number_of_groups
    if scipy.absolute(voltages).all() < EPSILON:
        conductance = scipy.nan
    else:
        conductance = (scipy.sum(scipy.absolute(
                       out_vec[n:n+self.input_count])) /
                      scipy.absolute(voltages))
    if hasattr(conductance, "__len__"):  # Dirty, but useful
        conductance = scipy.nan
    return (current_matrix, voltage_matrix, conductance)
```

Fortran

```
subroutine read_sensor_grid(sensor_grid, current_matrix, size, iw, ih, result)
    ! Read the sensor grid, averaging the current over all
    ! the tunnels that pass through each grid sensor.
    ! This is still the slowest part of the code, but being
    ! in Fortran certainly speeds it up.
    implicit none
    integer, parameter :: double = selected_real_kind(15)
    integer, parameter :: long = selected_int_kind(15)
    integer(kind=long), intent(in) :: size, iw, ih
    real(kind=double), dimension(iw, ih), intent(in) :: sensor_grid
    real(kind=double), dimension(iw, ih), intent(in) :: current_matrix

    real(kind=double), dimension(size), intent(out) :: result

    integer(kind=long) :: n, i, j, cnt, one
    real(kind=double) :: total

    one = 1

!$OMP PARALLEL DO PRIVATE(n, i, j, cnt, total)
    do n = 1, size
        total = 0
        cnt = 0
        do j = 1, ih
            do i = j, iw
                if ( sensor_grid(i, j) .eq. n ) then
                    total = total + current_matrix(i, j)
                    cnt = cnt + 1
                end if
            end do
        end do
        result(n) = total / max(one, cnt)
    end do
!$OMP END PARALLEL DO
end subroutine read_sensor_grid

subroutine g_matrix(conductance_matrix, current_matrix, gn, in, result)
    ! Generate the G matrix based on the conductance matrix
    ! and the skeleton "current_matrix".
    implicit none
    integer, parameter :: double = selected_real_kind(15)
    integer, parameter :: long = selected_int_kind(15)

    integer(kind=long), intent(in) :: gn, in
    real(kind=double), dimension(gn, gn), intent(in) :: conductance_matrix
    real(kind=double), dimension(in, in), intent(in) :: current_matrix

    real(kind=double), dimension(in, in), intent(out) :: result

    real(kind=double), dimension(gn) :: diagonals
    integer(kind=long) :: i

    diagonals = sum(conductance_matrix, dim=2)
    result = current_matrix
    result(1:gn, 1:gn) = conductance_matrix
!$OMP PARALLEL DO
    do i = 1, gn
        result(i, i) = -diagonals(i)
    end do
!$OMP END PARALLEL DO
end subroutine g_matrix

subroutine voltage_matrix(voltages, distances, n, result)
    ! Calculate the voltage drops across the network.
    ! If the distance across a jump is infinite,
    ! the voltage drop will be nothing
    ! (because conductance will have been zero)
    use ieee_arithmetic
    implicit none
    integer, parameter :: double = selected_real_kind(15)
    integer, parameter :: long = selected_int_kind(15)

    integer(kind=long), intent(in) :: n
    real(kind=double), dimension(n), intent(in) :: voltages
    logical, dimension(n, n), intent(in) :: distances

    real(kind=double), dimension(n, n), intent(out) :: result

    integer(kind=long) :: i, j

!$OMP PARALLEL DO
    do i = 1, n
        do j = i, n
            result(j, i) = abs(voltages(j) - voltages(i))
            result(i, j) = result(j, i)
        end do
    end do
!$OMP END PARALLEL DO
    where (distances) result = 0
end subroutine voltage_matrix

subroutine current_matrix(voltage_matrix, conductance_matrix, sizes, n, result)
    ! Calculate the currents in the network.
    ! If the size of a gap is infinite, the current
    ! will be zero because the conductance will be zero.
    use ieee_arithmetic
    implicit none
    integer, parameter :: double = selected_real_kind(15)
    integer, parameter :: long = selected_int_kind(15)

    integer(kind=long), intent(in) :: n
    real(kind=double), dimension(n, n), intent(in) :: voltage_matrix
    real(kind=double), dimension(n, n), intent(in) :: conductance_matrix
    logical, dimension(n, n), intent(in) :: sizes

    real(kind=double), dimension(n, n), intent(out) :: result

    result = voltage_matrix * conductance_matrix
    where (sizes) result = 0
end subroutine current_matrix
```

B Learner Parameters

The parameters below were used when the experiments were run. If a parameter is not applicable to a particular learner, it should not be considered present. For example, spectral radius does not apply to neuromorphic reservoirs.

Fostner and Brown Replications

Parameter	Current Spikes	Conductance Curves
Width	200	100
Height	200	100
Coverage	0.65	0.65
Tunnel	Switch	Switch
Voltage range	0 V to 1 V	0 V to 1 V
Voltage step size	0.025 V	0.0001 V
Voltage cycles	5 up, 5 down	1
Probability of switch-up	0.1	0.01, 0.1, 0.8
Probability of switch-down	0	0

MNIST Database

Parameter	Value
Input dimension	64
Output dimension	10
Reservoir size	100, 200, 500
Leaking rate	1
Regularisation	1×10^{-8}
Type	ESN, Memristor, Switch, Resistor
Spectral radius	0.5
Sparsity	0.75

Mackey-Glass Learners Test One

Parameter	Value
Input dimension	1
Output dimension	1
Reservoir size	100, 200, 500
Leaking rate	0.3
Regularisation	1×10^{-8}
Type	ESN, Memristor, Switch, Resistor
Spectral radius	0.5
Sparsity	0.75

Mackey-Glass Learners Test Two

Parameter	Value
Input dimension	1
Output dimension	1
Reservoir size	50, 100, 200, 500, 750, 1000, 1500, 2000
Leaking rate	0.5
Regularisation	1×10^{-8}
Type	Feed-forward ESN, Memristor, One-hop ESN
Spectral radius	1
Sparsity	0.2

Mackey-Glass Test Conditions

Parameter	Value
τ	17
n	10
β	0.2
γ	0.1

C Full Data Analysis

MNIST Database

In the following tables, precision is the left number and recall is the right number.

100-neuron learners

Digit	ESN		Memristors		Atomic Switches		Resistors	
0	0.9454	0.9773	0.9131	0.9943	0.8963	0.9841	0.9127	0.9909
1	0.8954	0.8802	0.8630	0.8736	0.8579	0.8297	0.9119	0.8681
2	0.9544	0.9372	0.9629	0.9105	0.9325	0.8942	0.9376	0.9256
3	0.8914	0.8363	0.9006	0.8319	0.8948	0.8264	0.9025	0.8187
4	0.9592	0.9109	0.9579	0.9000	0.9374	0.8685	0.9477	0.8783
5	0.8686	0.9011	0.8974	0.9352	0.8623	0.8692	0.8821	0.9132
6	0.9547	0.9824	0.9630	0.9824	0.9289	0.9868	0.9412	0.9769
7	0.9340	0.9191	0.9262	0.9303	0.9074	0.9191	0.9340	0.9315
8	0.8587	0.8034	0.8519	0.8591	0.8559	0.7886	0.8423	0.8295
9	0.7830	0.8717	0.8898	0.8902	0.7607	0.8446	0.8204	0.8826
Mean	0.9045	0.9020	0.9126	0.9108	0.8834	0.8811	0.9032	0.9015

200-neuron learners

Digit	ESN		Memristors		Atomic Switches		Resistors	
0	0.9515	0.9636	0.9321	0.9193	0.8954	0.9716	0.9389	0.9943
1	0.8940	0.8659	0.8503	0.7714	0.8469	0.7549	0.8687	0.8582
2	0.9622	0.9244	0.9304	0.7756	0.8892	0.8756	0.9469	0.8965
3	0.9017	0.8275	0.9003	0.7846	0.8086	0.7802	0.8983	0.8253
4	0.9445	0.8989	0.9476	0.8783	0.9294	0.8717	0.9445	0.8750
5	0.8639	0.9121	0.8987	0.8505	0.8360	0.8330	0.8841	0.9022
6	0.9372	0.9813	0.9504	0.9418	0.9179	0.9484	0.9410	0.9681
7	0.9065	0.8966	0.8892	0.8640	0.8911	0.8820	0.9223	0.9034
8	0.8453	0.8114	0.7231	0.8716	0.7945	0.7580	0.8161	0.8409
9	0.7981	0.8946	0.8568	0.7880	0.7217	0.8250	0.8152	0.8826
Mean	0.9005	0.8976	0.8879	0.8445	0.8531	0.8500	0.8976	0.8947

The 500-neuron table is in Chapter 5, page 37.