UNIVERSITY OF TENNESSEE, EECS DEPARTMENT

A Primer on Reservoir Computing

Aleksander Klibisz

December 8, 2016

COSC594, Unconventional Computation, Fall 2016, Dr. Bruce MacLennan

1 INTRODUCTION

This paper is a brief primer on concepts in reservoir computing (RC). Reservoir computing defines a class of artificial neural network (ANN) models that mimic neural microcircuits in the biological brain using an un-trained reservoir of neurons and a trained readout function. This structure distinguishes RC models from traditional ANNs which train neurons at all layers in the network. The remainder of this paper is organized as follows. Section 2 introduces fundamental concepts in ANNs necessary for understanding RC models. Section 3 introduces high-level concepts, motivations, and the historic development of RC. Section 4 covers the Liquid State Machine (LSM) model in a more formal capacity. Section 5 details several practical aspects of RC. Section 6 highlights recent work and future trends in RC research.

2 PRELIMINARIES

An introduction to several concepts in artificial neural networks is helpful to develop an understanding of RC. For brevity, only the concepts necessary to present reservoir computing and liquid state machines are covered.[1] Important terms are emphasized in bold font.

Artificial neural networks (ANNs) are loosely defined as a computational model that mimics the human brain in various capacities. ANNs take data on an **input layer**, apply transformations through a **hidden layer** of computational units (often called **neurons**), and return meaningful values on an **output layer**. This is commonly expressed by saying that ANNs are trained to learn $f(x) = y$, where x is the input data and y is a ground-truth property of that data.

An ANN can be **trained** by adjusting its computational behavior to learn a meaningful data representation through **supervised learning**, where the output layer should match explicit input labels, **unsupervised learning**, where the ANN learns an abstract representation without explicit feedback, and hybrids thereof.

[1] See http://neuralnetworksanddeeplearning.com for a much more thorough introduction.

ANNs have been a topic of research in artificial intelligence and machine learning since a mathematical model was proposed by McCulloch and Pitts (1943). Despite a theoretical foundation, their popularity stagnated over time due to lacking computational capacity for training. As computational resources improved, ANNs were leveraged for their ability to learn complex representations of data. Today they are a popular topic in academia and have been successfully implemented in industry for tasks such as computer vision, speech recognition, topic modeling, autonomous navigation, and others. The following sub-sections briefly introduce simple feed-forward neural networks, recurrent neural networks (which relate more closely to reservoir computing), and the difference between continuous activation neurons and spiking neurons.

2.1 Feed-Forward Neural Networks

A fundamental neural network model is the **Feed-Forward Neural Network** (FFNN). As the name suggests, the FFNN passes information in one direction from the input layer, through the hidden layer, and out of the output layer. For example, a FFNN trained for image classification might take an image defined as a vector of its numerical pixel intensities on the input layer, transform these values through the hidden layer, and return a probability distribution over its possible classifications on the output layer.

Neurons in the hidden layer process a weighted aggregate of their inputs to produce a single output that is passed onto subsequent neurons. The simplest example of a neuron is the **simple perceptron**, introduced by Rosenbaltt (1957). Fig. 2.1 shows a perceptron that transforms multiple binary inputs to a single output. The neuron has a numerical **weight** for each of these inputs and a single numerical **bias**. As expressed in Eq. (2.1), if the weighted sum of these inputs and the bias exceeds a certain threshold, the output is 1; otherwise the output is 0. Fig. 2.1 also shows that multiple perceptrons can be arranged to form a logical `NAND` operator, indicating this simple construct is universal for computation (Nielsen, 2015).

$$\text{output} = \begin{cases} 0 \text{ if } \sum_j w_j x_j + b \leq 0 \\ 1 \text{ if } \sum_j w_j x_j + b > 0 \end{cases} \tag{2.1}$$

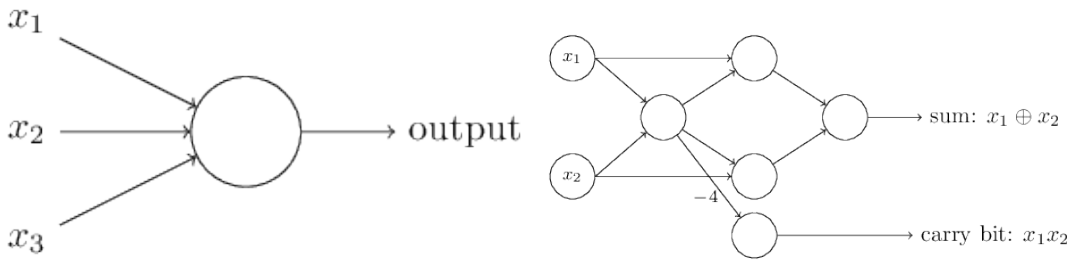

Figure 2.1: (Left) a single perceptron; (right) perceptrons arranged to form `NAND`, each perceptron has weight −2 and bias 3 unless otherwise indicated. (Nielsen, 2015)

Layers in an ANN are comprised of perceptrons, each characterized by its weights w_i corresponding to inputs x_i and a single bias b. The inputs are passed through the layers using **forward propagation** (FP), commonly expressed as a matrix multiplication at each layer. It turns out the simple perceptron's weighted sum performs poorly with common learning algorithms, so it is generally replaced by other **activation functions**, such as the continuous **sigmoid function** per Eq. (2.2).

$$\text{output} = \sigma(z) = \frac{1}{1+e^{-z}}, \text{ where } z = w \cdot x + b \tag{2.2}$$

In training, the neural network's weights and biases[2] are adjusted to minimize an **objective function**, such as the **mean-squared error** measure. The objective function serves to evaluate the performance of the current network weights and biases for every input-output pair (x, y).

Back propagation (BP) in combination with an optimization technique like **gradient descent** (GD) comprise a common algorithm for adjusting the neuron weights and biases to minimize the objective function. BP computes a FP through the ANN for one or more input-output pairs and then works backwards through the network to compute a **gradient** at each layer. The gradient is roughly interpreted as the amount by which each layer contributed to the cost for that input-output pair. GD uses the gradient to adjust weights and biases for the next FP. This is continued for a fixed number of passes over all input-output pairs or until the cost converges.

2.2 Recurrent Neural Networks

FFNNs as previously described operate under the assumption that input-output pairs are independent of one another. This works well for some types of data (e.g. a series of un-related images), but fails to capture intrinsic sequential or temporal dependencies in other contexts (e.g. time-series analysis, audio processing, generative topic modeling). **Recurrent neural networks** (RNNs) capture this temporal information using cyclic connections in the hidden layer that can preserve internal state between input evaluations (Karpathy, 2015).

A simple way to express the distinction between the FFNN model and the RNN model is that the FFNN operates as a function $f(x_t) = y_t$, whereas the RNN operates as a dynamical system $f(x_t, s_{t-1}) = y_t, s_t$. That is, the FFNN output is a function of its input, and the RNN output at time t is a function of both the input x_t and the prior internal state s_{t-1} (Britz, 2015).

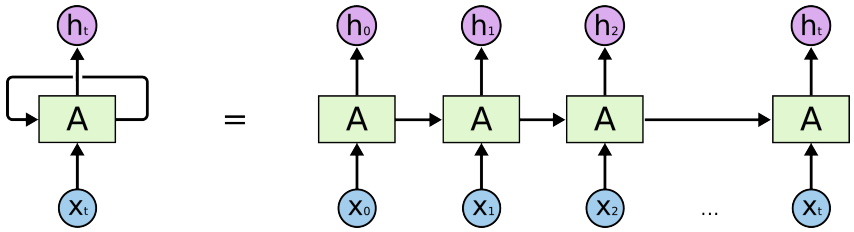

Figure 2.2: A symbolic RNN that computes $f(x_t) = h(t)$ with an interal loop can be unrolled into a series of states (Olah, 2015).

Standard back propagation is not sufficient for RNNs due to cycles in the hidden layer. Thus, the **Back Propagation Through Time** algorithm (BPTT), was introduced by Werbos (1990) to train RNNs. As shown in Fig. 2.2, the RNN loops can be unrolled into a series of states. BPTT first does this unrolling and then proceeds with BP, maintaining the prescribed sequence of input-output pairs.

Some extensions of RNNs have been designed to overcome shortcomings of standard RNNs. For example, standard RNNs have trouble preserving temporal dependencies as the amount of intermediate time grows. A standard RNN used for generative text modeling may "forget" the first word of a long sentence when sampling the last word of the sentence. This issue is addressed by an extension of RNNs called **Long-Short Term Memory** RNNs (LSTMs). Introduced by Hochreiter and Schmidhuber (1997), LSTMs consist of cells with input gates, output gates, and forget gates. This allows for more granular control over state and longer preservation of state in the network.

[2] Some implementations hold the bias constant instead of adjusting.

2.3 Continuous Activation Neurons vs. Spiking Neurons

ANNs can be constructed from **continuous activation neurons** or **spiking neurons**. Continuous activation neurons are characterized as having an activation function that is evaluated over its input, and the resulting value is always passed to the next layer of neurons. For example, FFNNs may use neurons with a sigmoid activation function. For every set of inputs, these neurons will compute the sigmoid function on the inputs and pass the result to the next layer of neurons. In contrast, a spiking neuron will accumulate values from its inputs up to a certain threshold. Only when this threshold is reached does the spiking neuron pass a value to the next layer of neurons. Maass (1997) received significant credit for formalizing this concept. Their work demonstrated that spiking neurons can compute any function computed by sigmoid neurons using fewer neurons and observed that spiking neurons more closely resemble the human brain.

3 Introduction to Reservoir Computing

This section provides a high-level look at the motivation, structure, inspiration, history, distinct models, advantages, and disadvantages of reservoir computing.

3.1 Motivation

The motivation for RC is driven by the need to process temporal information and the computational expense of training existing models (e.g. RNN, LSTM) to handle this type of information. To this end, reservoir computing seeks to be a high-quality alternative with lower computational expense in training.

3.2 Structural Characteristics

The RC model is characterized as having two key components: the **reservoir** and the **readout function**, both depicted in Fig. 3.1. The reservoir consists of randomly-initialized, recurrently-connected (i.e. with cycles) neurons, corresponding roughly to the hidden layer of traditional ANNs. Unlike traditional ANN implementations, this part of the model is intentionally left untrained. Inputs are passed from an input layer and through the reservoir. As information exits the reservoir, the readout function is used to glean a meaningful output representation. To achieve this output, the readout function is implemented as a simpler model and trained to make sense of the reservoir outputs. The readout function is typically conceptually simple and computationally inexpensive to train. An example is linear regression, where the model learns a set of weights corresponding to each of the reservoir outputs.

3.3 Inspirations

Some may find the idea of using an untrained pool of neurons counter-intuitive when compared to other ANNs that rely on very precise learned weights and neuron-level configurations. The inspiration for this concept is commonly attributed to knowledge of biological brains, and work in this area frequently refers to theoretical and empirical knowledge of brains. For example, Maass and Markram (2004) began their introduction to the LSM model by emphasizing empirical evidence that cortical microcircuits in the human brain are not re-trained or adjusted for specific tasks, but rather maintain the same configuration as they are re-arranged for different tasks in different parts of the brain. In one of the first demonstrations of LSMs for a benchmark application, Joshi and Maass (2004) configured neurons and synapses based on knowledge of rat brains for an LSM used to maneuver a simulated

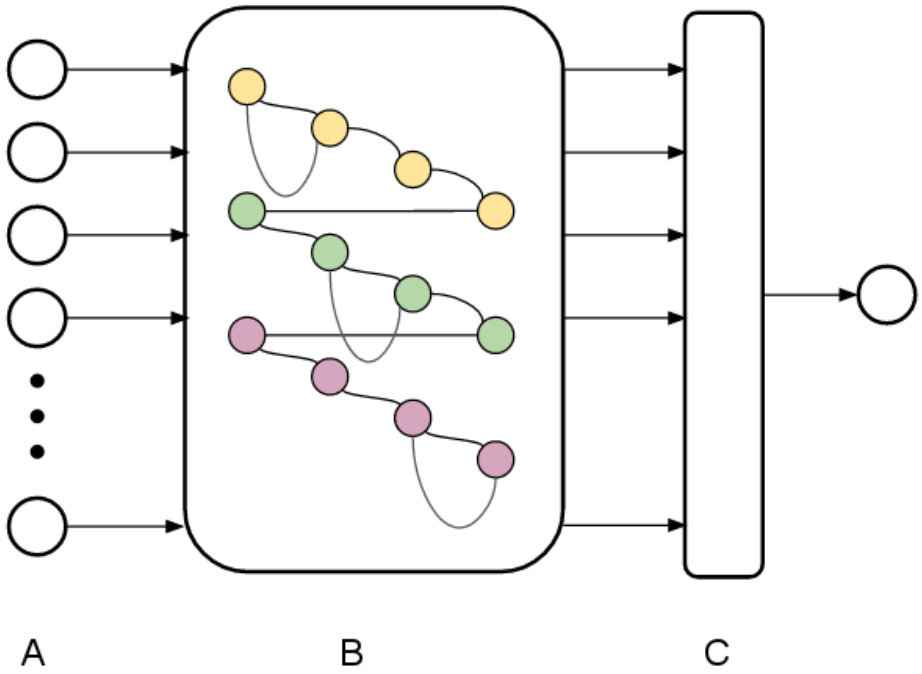

Figure 3.1: A conceptual expression of the reservoir computing model. (A) the input layer. (B) the **reservoir** is implemented as an un-trained, randomly-initialized pool of recurrently-connected neurons. (C) the **readout function** is implemented as a simpler machine learning or statistical model (e.g. linear regression) and is trained to provide meaningful output.

robot arm. This close connection to neurological anatomy is exemplified in the case of researcher Peter F. Dominey. Dominey spent roughly a decade studying circuits in the human brain for their role in language learning before becoming aware of RC and its many similarities to his work; the similarities are documented in Dominey (2013).

3.4 History and Distinct Models

Reservoir computing has a relatively brief history, with the first formal models being introduced in the early 2000s. Jaeger et al. (2007) provides a concise summary as of 2007. The idea of combining randomly-initialized pools of neurons was mentioned by Rosenblatt (1961) and Hinton (1981). However, Buonomano and Merzenich (1995) first formulated the concept in a way that closely resembles its modern form. Specifically, they modeled a random network of spiking neurons with properties based on empirical knowledge of the brain. When combined with a trained readout layer, the model was able to accurately distinguish several temporal patterns.

Jaeger (2001) is credited with introducing the **Echo State Network** (ESN), which was described as "a constructive learning algorithm for recurrent neural networks, which modifies only the weights to output units in order to achieve the learning task." Shortly thereafter, Maass et al. (2002) introduced **Liquid State Machines** (LSM), similarly described as a method for computing time-varying input without task-dependent construction of neural microcircuits. Steil (2004) devised a new learning scheme for RNNs called **Backpropagation-Decorrelation** (BPDC) that utilizes non-adaptive neurons based specifically on ESNs and LSMs. Verstraeten et al. (2007) first proposed to merge the research of ESNs, LSMs, and BPDC under the unified name *Reservoir Computing*.

Two other models appearing less frequently in RC literature are **Evolino** (Evolution of systems with linear outputs) (Schmidhuber et al., 2007) and **Extreme Learning Machines** (Huang et al., 2006).

3.5 Advantages

1. **Minimized Training Cost** - an untrained, randomly-initialized pool of neurons can achieve results comparable to a trained recurrent neural network without the computational expense and complexity of training the recurrent layers of neurons.
2. **Inherent Parallelism** - a single reservoir may be connected to multiple readout functions, each trained to serve a different purpose (Maass and Markram, 2004).
3. **Biologically Accurate Representation** - the RC structure models known properties of biological brains more accurately than many other popular implementations of ANNs. For cases where this is considered advantageous, RC models are favorable.

3.6 Disadvantages

1. **Task-specific Reservoir Tuning** - literature indicates little consensus on task-agnostic methods to determine the best reservoir for a given task. While RC models have performed well on benchmarks, the use of random reservoirs cannot guarantee optimal performance. Some progress has been made to this end and will be covered in Section 5.1.

4 Liquid State Machines

The Liquid State Machine (LSM) is as an implementation of the RC paradigm introduced by Maass et al. (2002). This section briefly introduces the formal model definition, requirements imposed for computational utility, and a comparison to the Echo State Network model (ESN). Unless otherwise indicated, the theory and definitions presented in this section are attributed to Maass et al. (2002), Maass and Markram (2004), and Maass (2010).

4.1 Formal Model Definition

The LSM is defined in terms of **filters**, where a filter is a mapping between two functions of time. As depicted in Fig. 4.1, an LSM $M = \langle L^M, f^M \rangle$ consists of a liquid filter L^M and the memory-less readout function f^M. The LSM M computes a filter from input function $u(\cdot)$ to $y(t)$. The components and their semantics are further broken-down as follows.

Definition 4.1. Liquid State Machine $M = \langle L^M, f^M \rangle$
The LSM M is comprised of a liquid filter L^M and a readout function f^M.

Definition 4.2. Input function $u(\cdot)$
Sometimes denoted $u(s)$ for all $s \leq t$. This is a function of time that serves as input to the LSM. Its notation should indicate that the eventual output function $y(t)$ is dependent upon more than just the value of u at time t, but rather potentially many values of u for times $s \leq t$.

Definition 4.3. Liquid filter L^M
This filter is comprised of finitely many basis filters from class **B**, which should satisfy the point-wise separation property. Often referred to as the *reservoir*, it serves as an information pre-processing layer that transforms its input into a higher-dimensional dynamical system.

Definition 4.4. Liquid state $x^M(t) = (L^M u)(t)$
The state of the LSM at time t is the result of applying the liquid filter L^M to the input function u at time t.

Definition 4.5. Memory-less readout function f^M
This is a single function taken from a pool **R** which should satisfy the universal approximation property. It is trained (most commonly in a supervised fashion) to take input from the liquid filter and deduce a meaningful output.

Definition 4.6. Output function $y(t) = (Mu)(t) = f^M(x^M(t))$
This is the output of the LSM, defined at time t as the result of applying the readout function f^M to the liquid state $x^M(t)$, equivalently defined as the result of applying LSM M to input function u at time t.

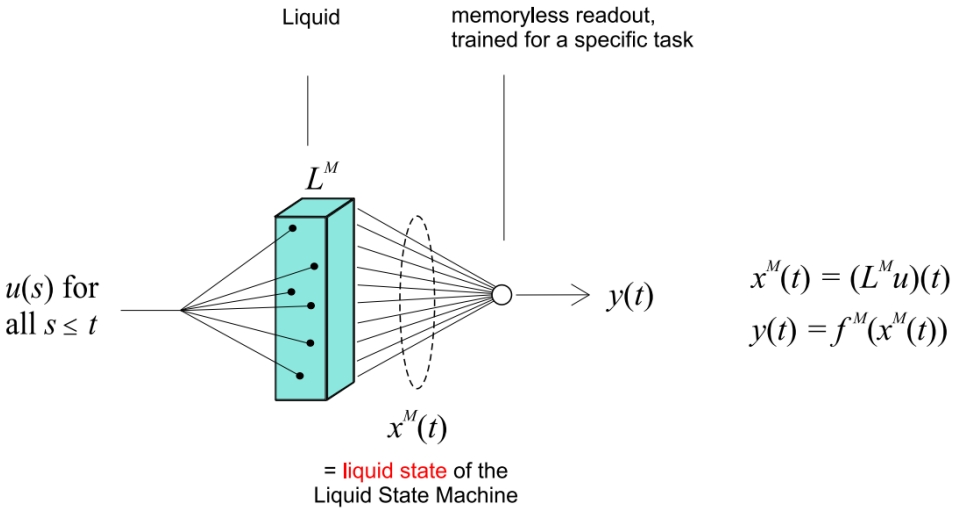

Figure 4.1: Components of the LSM from Maass (2010)

4.2 Computational Capacity and Requirements

4.2.1 Time-invariant, Fading-memory Filters

Maass et al. (2002) presents the LSM as a means for computing a particularly useful and large class of filters called **time-invariant, fading-memory filters**, defined as follows.

Definition 4.7. Time-invariant Filter
A filter F is time-invariant if a temporal shift in its input function by some amount causes a shift in the output function by the same amount.

Definition 4.8. Fading-memory Filter
A filter F has fading-memory if its output function depends on a finite time interval of its input function going into the past. That is, a fading memory filter depends less and less on its initial state as time passes and will eventually effectively "forget" its initial state.

The liquid filter and readout function of an LSM should satisfy two requirements to implement time-invariant, fading-memory filters. These requirements are expressed formally as Theorem 3.1 of Maass and Markram (2004) and Theorem 1.1 of Maass (2010); they are paraphrased in Theorem 1.

Theorem 1. *Paraphrasing Theorem 3.1 Maass and Markram (2004)*
In order for an LSM $M = \langle L^M, f^M \rangle$ to implement a time-invariant, fading-memory filter, two requirements are imposed:

1. *Filters in L^M satisfy the point-wise separation property, (Definition 4.9).*
2. *Readout function f^M satisfies the universal approximation Property, (Definition 4.10).*

Definition 4.9. Point-wise Separation Property
Class CB of filters has the point-wise separation property with regard to input function $u(\cdot) \in U^n$ if for any two functions $u(\cdot), v(\cdot) \in U^n$ with $u(s) \neq v(s)$ for some $s \leq 0$, there exists some filter $B \in CB$ such that $(Bu)(0) \neq (Bv)(0)$. In other words, there should exist a filter that can distinguish two distinct input functions from one another at the same time.

Definition 4.10. Universal Approximation property
Class CF of functions has the universal approximation property if for any $m \in N$, any set $X \subseteq \mathbb{R}^m$, any continuous function $h : X \mapsto \mathbb{R}$, and any given $\rho > 0$, there exists some function $f \in CF$ such that $|h(x) - f(x)| \leq \rho$ for all $x \in X$. In other words, any continuous function on a compact domain can be uniformly approximated by functions from CF.

One need not look far to find components that satisfy these requirements. Some examples of filters satisfying the point-wise separation property are linear filters with impulse responses, delay filters, leaky-integrate-and-fire neurons, threshold-logic gates, and even a pool of water. Examples of readout functions satisfying the universal approximation property include simple linear regression, ANN perceptrons, and support vector machines (Schrauwen et al., 2007).

4.2.2 Universal Computation

The computational capacity of LSMs expands beyond the class of time-invariant, fading memory filters into the realm of universal computation. Specifically, theorem 1.2 in Maass (2010) states that the original LSM structure can be augmented with a feedback function K to achieve universal computational capacity for analog computing. This feedback function K transfers output from a readout function back into the neural circuit, as depicted in Fig. 4.2. Based on this theorem, there exist combinations of a finite number of feedback functions that can simulate dynamical systems, which can simulate arbitrary Turing machines, making a properly-configured LSM universal for both analog and digital computation.

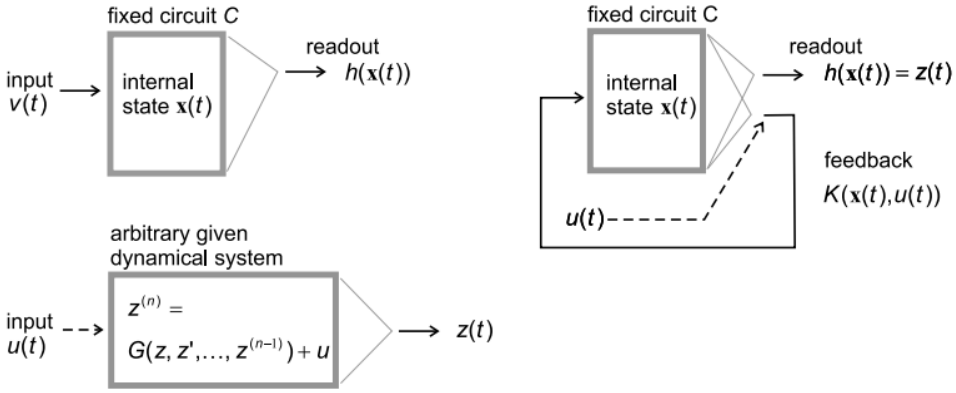

Figure 4.2: (Right) The LSM augmented with a feedback function K towards universal computation (Maass, 2010).

4.3 Comparison to Echo State Networks

The Echo State Network introduced by Jaeger (2001) closely resembles the LSM in structure and purpose. The theoretical formulation of LSMs is in fact general enough to also encompass ESNs (Jaeger et al., 2007). The primary difference in the two models is the implementation of neurons in the reservoir. While LSMs are generally implemented using spiking neurons, ESNs utilize simpler continuous activation neurons (Maass, 2010).

5 Reservoir Computing in Practice

This section highlights three practical components of RC: constructing the reservoir, choosing and training the readout function, and several open-source software simulators for RC simulation.

5.1 Constructing the Reservoir

When approaching a specific task with reservoir computing it is necessary to define configurations for the processing units in the reservoir, their topology, and their connectivity. Working towards a task-agnostic approach is considered an important topic in RC research.

Verstraeten et al. (2007) highlighted discrepancies in several common measurements for reservoir quality across several tasks and demonstrated an analytical approach to evaluating neuron and synapse properties across three tasks with the goal of characterizing a performant reservoir. They reported consistent indicators of performance in the reservoir size, the reservoir's spectral radius, and a newly-proposed pseudo-Lyapunov exponent. Haeusler and Maass (2007) similarly characterized the performance of cortical microcircuit models using statistical analysis. LukošEvičIus and Jaeger (2009) surveyed the state of techniques for both reservoir and readout function configuration. The survey classifies techniques for training reservoirs into three classes: using task-agnostic measurements to construct a reservoir, using unsupervised learning to adjust the reservoir with respect to the given input, and using supervised training to adjust the reservoir with respect to both the input and desired output. All methods have been explored to some extent in literature, and LukošEvičIus and Jaeger (2009) serves as a particularly apt reference for understanding the nuance of each.

For specific examples, consider some applications of RC and their reported methods of reservoir construction:

1. Jaeger and Haas (2004) leveraged ESNs for a benchmark time-series prediction task with an improvement factor of 2400 over the next best method. They used a reservoir of 1000 neurons with 1% connectivity. As part of their reasoning they stated: "This condition lets the reservoir decompose into many loosely coupled subsystems, establishing a richly structured reservoir of excitable dynamics."
2. Joshi and Maass (2004) trained an LSM to control a two-joint robotic arm. The reservoir included 600 leaky-integrate-and-fire neurons in a 20x5x6 configuration. Connections were sampled at random with a bias towards local connectivity. Neuron and synapse parameters (e.g. neuron threshold for passing a signal) were chosen based on knowledge of a rat brain.
3. Butcher et al. (2013) trained a hybrid RC / Extreme Learning Machine model for time-series prediction using a manual grid-search over reservoir parameters.

5.2 Choosing and Training the Readout Function

The readout function is generally implemented as a well-understood model from other areas of statistics and machine-learning. In their review of options for the readout function, LukošEvičIus and Jaeger (2009) describe successful implementation of single-layer readouts (e.g. linear regression, support vector machines, support vector regression), multi-layer readouts (e.g. multi-layer perceptrons), readouts with learned delays for time-coded outputs, multiple readouts used in combination, and readouts that partially feed outputs back in as inputs. For training the readout function, traditional methods (e.g. direct analytical solutions, stochastic gradient descent) as well as more unconventional methods (e.g. evolutionary algorithms) have been employed.

Still, there seems to be no universal method for choosing the best readout function given a specific task. With such a wide variety of proven options, I posit a pragmatic approach for a researcher in RC would be to begin with the simplest or most familiar option and follow a path of intuition- and literature-driven trial-and-error thereafter.

5.3 Software Simulators

Efficient and accurate implementation of RC models, particularly the detailed behavior and connectivity of processing units in the reservoir, is for most a non-trivial software engineering task. To this end, several open-source simulators have been implemented and utilized in RC literature.

1. PCSIM, *A Parallel Neural Circuit Simulator* (Pecevski et al., 2009) is a distributed, multi-threaded simulator for heterogeneous networks. The software is written in C++ with Python bindings. It encompasses `pylsm`, a Python simulator specific to the LSM approach to RC.
2. Aureservoir (Holzmann, 2008) is a simulator for the ESN approch to RC. The software is written in C++ with Python bindings.
3. NEST (Eppler et al., 2015), (Eppler et al., 2009) is a simulator for a variety of spiking neural network models. It is not specific to RC, but may be used to implement RC models. The software is written in C++ with Python bindings.
4. OGER Engine, *Organic Environment for Reservoir Computing*, (Verstraeten et al., 2012), (Lukoševičius et al., 2012) is a Python software toolkit built for processing large sequential datasets with techniques in RC as well as other machine learning techniques.

6 Current and Future Research

This section highlights a hand-full of successful applications of RC in literature and offers discussion for future trends in RC research.

6.1 State of the Art

Since its start in the early 2000s, successful applications of RC include but are not limited to speech and handwriting recognition, robot motor control, time-series prediction, and medical brain-computer interfacing. Jaeger et al. (2007), Schrauwen et al. (2007), Maass (2010), and Lukoševičius et al. (2012) provide thorough surveys of applications that depict how RC has grown over time. To supplement these articles, several contributions in RC since 2012 are highlighted.

1. Paquot et al. (2012) implemented an opto-electronic hardware architecture for RC and demonstrated its utility on channel equalization and speech recognition tasks.

2. Butcher et al. (2013) overcame a limitation of short-term memory in ESNs by combining concepts from RC and Extreme Learning Machines. The hybrid model was tested on a highly nonlinear time-series prediction task.
3. Salehi et al. (2014) used a simulated network of photonic crystal cavities to implement a reservoir. The system reached 100% accuracy on a benchmark voice-recognition dataset.
4. Antonik et al. (2015) implemented a reservoir and readout function on a field-programmable gate array (FPGA) with online-learning. The implementation was tested on a nonlinear communication channel equalisation task; results on hardware matched those in software simulation.
5. Alomar et al. (2016) implemented an LSM with stochastic spiking neurons in a ring topology on an FPGA. The implementation was tested with a time-series prediction task.

6.2 Research Trends

Based on the works surveyed for this paper, my speculative interpretation of the RC field sees research trending in three distinguishable areas: physical implementations, practical application, and task-agnostic methodology for finding performant reservoirs.

6.2.1 Physical Implementations

It appears a considerable portion of the work since 2013 has focused on analog, optical, and FPGA implementations of existing RC paradigms. These implementations are promising in that they exploit speed and parallelism that cannot be obtained on digital computers. I see this trend continuing as there is still room to grow in terms of building general-purpose hardware and demonstrating real-world utility beyond benchmark datasets. Lukoševičius et al. (2012) expressed a similar sentiment in saying "Thus RC has spread well beyond the world of artificial neural networks. In particular, it enables useful computation on hardware platforms where it is hard to implement, e.g., basic electronic equivalents of logic gates and memory cells. Potential and functioning examples include analog electronics, randomly crystallized nonlinear electronic networks, opto-electronic and optical systems, or just a bucket filled with water."

6.2.2 Practical Application

RC implementations have matched or exceeded the performance of conventional RNNs on multiple benchmark tasks dealing with temporal data. Still, it seems that LSTMs have so far been a more popular choice for wide-spread application. As an example, Google chose LSTMs as a replacement to Gaussian Mixture Models for voice-to-text transcription in 2015[3]. RC stands to benefit if its implementation can demonstrate utility beyond a research environment.

6.2.3 Task-Agnostic Methodology for Tuning Reservoirs

It seems a comprehensive, task-agnostic methodology for finding performant reservoirs has eluded RC researchers thus far. Publications commonly report a methodology constrained to mimicry of known neurological properties or brute-force search for optimal parameters. If RC computing has come this far with a foundation of randomness and convention, a more precise understanding and methodology for configuring RC implementations would surely reap benefits for researchers and practitioners.

[3] https://research.googleblog.com/2015/08/the-neural-networks-behind-google-voice.html

7 Conclusion

Reservoir computing is a class of artificial neural network characterized by a randomly-initialized reservoir of recurrently-connected neurons feeding into a simpler trained readout function. This model has shown particular promise for tasks involving temporal dependencies. The relatively low computational expense of training RC models is an appealing advantage over more traditional recurrent neural networks. This work presented a high-level introduction and summary of concepts in RC. A preliminary introduction to ANNs was presented to cover prerequisite concepts. An introduction to RC included the motivation, structural characteristics, inspirations, and history of RC. The liquid state machine model was introduced in a formal capacity. Techniques for constructing the reservoir and readout function and tools for software simulation were briefly surveyed. Finally, several successful applications from the last four years were highlighted, and predictions for future research were defined.

REFERENCES

M. L. Alomar, V. Canals, A. Morro, A. Oliver, and J. L. Rossello. Stochastic hardware implementation of liquid state machines. In *Neural Networks (IJCNN), 2016 International Joint Conference on*, pages 1128–1133. IEEE, 2016.

P. Antonik, A. Smerieri, F. Duport, M. Haelterman, and S. Massar. Fpga implementation of reservoir computing with online learning. In *24th Belgian-Dutch Conference on Machine Learning*, 2015.

D. Britz. Introduction to rnns, sep 2015. URL `http://www.wildml.com/2015/09/recurrent-neural-networks-tutorial-part-1-introduction-to-rnns/`.

D. V. Buonomano and M. M. Merzenich. Temporal information transformed into a spatial code by a neural network with realistic properties. *Science*, 267(5200):1028, 1995.

J. Butcher, D. Verstraeten, B. Schrauwen, C. Day, and P. Haycock. Reservoir computing and extreme learning machines for non-linear time-series data analysis. *Neural networks*, 38:76–89, 2013.

P. F. Dominey. Recurrent temporal networks and language acquisitionâĂŤfrom corticostriatal neurophysiology to reservoir computing. 2013.

J. M. Eppler, M. Helias, E. Muller, M. Diesmann, and M.-O. Gewaltig. Pynest: a convenient interface to the nest simulator. 2009.

J. M. Eppler, R. Deepu, C. Bachmann, T. Zito, A. Peyser, J. Jordan, R. Pauli, L. Riquelme, S. Albada, A. Morrison, et al. Nest 2.8. 0. Technical report, JARA-HPC, 2015.

S. Haeusler and W. Maass. A statistical analysis of information-processing properties of lamina-specific cortical microcircuit models. *Cerebral cortex*, 17(1):149–162, 2007.

G. E. Hinton. Implementing semantic networks in parallel hardware. *Parallel models of associative memory*, pages 161–187, 1981.

S. Hochreiter and J. Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

G. Holzmann. aureservoir, 2008. `http://mloss.org/software/view/138/`.

G.-B. Huang, Q.-Y. Zhu, and C.-K. Siew. Extreme learning machine: theory and applications. *Neurocomputing*, 70(1):489–501, 2006.

H. Jaeger. The âĂIJecho stateâĂİ approach to analysing and training recurrent neural networks-with an erratum note. *Bonn, Germany: German National Research Center for Information Technology GMD Technical Report*, 148:34, 2001.

H. Jaeger and H. Haas. Harnessing nonlinearity: Predicting chaotic systems and saving energy in wireless communication. *science*, 304(5667):78–80, 2004.

H. Jaeger, W. Maass, and J. Principe. Special issue on echo state networks and liquid state machines. *Neural Networks*, 20(3):287–289, 2007.

P. Joshi and W. Maass. Movement generation and control with generic neural microcircuits. In *International Workshop on Biologically Inspired Approaches to Advanced Information Technology*, pages 258–273. Springer, 2004.

A. Karpathy. The unreasonable effectiveness of recurrent neural networks, may 2015. URL `http://karpathy.github.io/2015/05/21/rnn-effectiveness/`.

M. LukošEvičIus and H. Jaeger. Reservoir computing approaches to recurrent neural network training. *Computer Science Review*, 3(3):127–149, 2009.

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

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

W. Maass. Liquid state machines: motivation, theory, and applications. *Computability in context: computation and logic in the real world*, pages 275–296, 2010.

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

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

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

M. A. Nielsen. Neural networks and deep learning. *URL: http://neuralnetworksanddeeplearning. com/.(visited: 01.11. 2014)*, 2015.

C. Olah. Understanding lstm networks, aug 2015. URL `http://colah.github.io/posts/2015-08-Understanding-LSTMs/`.

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

D. Pecevski, T. Natschläger, and K. Schuch. Pcsim: a parallel simulation environment for neural circuits fully integrated with python, 2009.

F. Rosenbaltt. The perceptron–a perciving and recognizing automation. Technical report, Report 85-460-1 Cornell Aeronautical Laboratory, Ithaca, 1957.

F. Rosenblatt. Principles of neurodynamics. perceptrons and the theory of brain mechanisms. Technical report, DTIC Document, 1961.

M. Salehi, E. Abiri, and L. Dehyadegari. Nanophotonic reservoir computing for noisy time series classification. *International Journal of Computer and Electrical Engineering*, 6(3):240, 2014.

J. Schmidhuber, D. Wierstra, M. Gagliolo, and F. Gomez. Training recurrent networks by evolino. *Neural computation*, 19(3):757–779, 2007.

B. Schrauwen, D. Verstraeten, and J. Van Campenhout. An overview of reservoir computing: theory, applications and implementations. In *Proceedings of the 15th European Symposium on Artificial Neural Networks. p. 471-482 2007*, pages 471–482, 2007.

J. J. Steil. Backpropagation-decorrelation: online recurrent learning with o (n) complexity. In *Neural Networks, 2004. Proceedings. 2004 IEEE International Joint Conference on*, volume 2, pages 843–848. IEEE, 2004.

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

D. Verstraeten, B. Schrauwen, S. Dieleman, P. Brakel, P. Buteneers, and D. Pecevski. Oger: modular learning architectures for large-scale sequential processing. *Journal of Machine Learning Research*, 13(Oct):2995–2998, 2012.

P. J. Werbos. Backpropagation through time: what it does and how to do it. *Proceedings of the IEEE*, 78(10):1550–1560, 1990.