12-2006

A Resource Efficient Localized Recurrent Neural Network Architecture and Learning Algorithm

Daniel Borisovich Budik
University of Tennessee - Knoxville

Recommended Citation

Budik, Daniel Borisovich, "A Resource Efficient Localized Recurrent Neural Network Architecture and Learning Algorithm. " Master's Thesis, University of Tennessee, 2006.
https://trace.tennessee.edu/utk_gradthes/1513

To the Graduate Council:

I am submitting herewith a thesis written by Daniel Borisovich Budik entitled "A Resource Efficient Localized Recurrent Neural Network Architecture and Learning Algorithm." I have examined the final electronic copy of this thesis for form and content and recommend that it be accepted in partial fulfillment of the requirements for the degree of Master of Science, with a major in Electrical Engineering.

Itamar Elhanany, Major Professor

We have read this thesis and recommend its acceptance:

Donald Bouldin, Gregory Peterson

Accepted for the Council:
Carolyn R. Hodges

Vice Provost and Dean of the Graduate School

(Original signatures are on file with official student records.)

To the Graduate Council:
I am submitting herewith a thesis written by Daniel Borisovich Budik entitled "A Resource-Efficient Localized Recurrent Neural Network Architecture and Learning Algorithm." I have examined the final electronic copy of this thesis for form and content and recommend that it be accepted in partial fulfillment of the requirements for the degree of Master of Science, with a major in Electrical Engineering.

Itamar Elhanany
Major Professor

We have read this thesis
and recommend its acceptance:

Donald Bouldin

Gregory Peterson

Accepted for the Council:

Linda Painter
Interim Dean of Graduate Studies

(Original signatures are on file with official student records.)

A Resource-Efficient Localized Recurrent Neural Network Architecture and Learning Algorithm

A Thesis

Presented for the Master of Science Degree

The University of Tennessee, Knoxville

Daniel Borisovich Budik

December 2006

Dedication

To my parents, Boris Budik and Bella Budik, and to my grandmother, Maria Shusterman. You raised me, supported me, taught me and loved me. This thesis is dedicated you.

Acknowledgments

It gives me great pleasure to thank the many people who made this thesis possible.

First and foremost, I would like to express gratitude to all of my math and science teachers from Webb School for opening up a window for me and making me realize that math and science can be fun. I would like to thank all of my undergraduate and graduate teachers at The University of Tennessee who have helped me along the way and provided me with the insight into what it means to be a computer engineer.

I am deeply indebted to my mentor and advisor Dr. Itamar Elhanany whose help, stimulating suggestions and encouragement continually steered me in my research and in the writing of this thesis.

I wish to thank numerous student colleagues at the The University of Tennessee for creating a stimulating and fun environment in which to learn and grow. I would like to especially thank Kenny Gilbert and Jared Pendleton from the Laboratory for Information Technologies (LIT) for helping me get through my undergraduate years. From the Machine Intelligence Laboratory (MIL), I am grateful to Brad Matthews for his patience and help with my design struggles and to Zhenzhen Liu for her insights in the mathematical aspects of my research.

Lastly, and most importantly I would like to thank my family for believing in me and offering me all of their support.

Abstract

Recurrent neural networks (RNNs) are widely acknowledged as an effective tool that can be employed by a wide range of applications that store and process temporal sequences. The ability of RNNs to capture complex, nonlinear system dynamics has served as a driving motivation for their study. RNNs have the potential to be effectively used in modeling, system identification and adaptive control applications, to name a few, where other techniques fall short. Most of the proposed RNN learning algorithms rely on the calculation of error gradients with respect to the network weights. What distinguishes recurrent neural networks from static, or feedforward networks, is the fact that the gradients are time-dependent or dynamic. This implies that the current error gradient does not only depend on the current input, output and targets, but rather on its possibly infinite past. How to effectively train RNNs remains a challenging and active research topic.

This thesis introduces TRTRL, an efficient, low-complexity online learning algorithm for recurrent neural networks. The approach is based on the real-time recurrent learning (RTRL) algorithm, whereby the sensitivity set of each neuron is reduced to weights associated either with its input or output links. As a consequence, storage requirements are reduced from $O(N^3)$ to $O(N^2)$ and the computational complexity is reduced from $O(N^4)$ to $O(N^2)$. Despite the radical reduction in resource requirements, it is shown through simulation results that the overall performance degradation of the truncated real-time recurrent learning (TRTRL) algorithm is minor. Moreover, the scheme lends itself to efficient hardware realization by virtue of the localized property that is inherent to the approach. The TRTRL algorithm is first implemented and evaluated using a multi-purpose CPU. Next, the framework is extended to a hardware implementation that scales to high network densities without compromising computation speed and overall performance.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 An Introduction to Recurrent Neural Networks

Artificial recurrent neural networks (RNNs) are widely acknowledged as an effective tool that can be used by a wide range of applications that store and process temporal sequences. The ability of RNNs to capture complex, nonlinear system dynamics has served as a driving motivation for their study. RNNs have the potential to be effectively used in a wide range of modeling, system identification and control applications, where other techniques may fall short. Consequently, a variety of learning algorithms have been proposed, the majority of which rely on the calculation of error gradients with respect to the network weights. What distinguishes recurrent neural networks from static, or feedforward networks, is the fact that the gradients are time-dependent or dynamic. This implies that the current error gradient does not only depend on the current input, output and targets, but rather on its possibly infinite past. How to effectively train RNNs remains a challenging and an active research topic.

1.2 Fundamentals

RNNs are neural networks which utilize recurrent links in order to provide dynamic memory. The recurrent connections allow the network's hidden units to see its own previous output, so that future behavior can be shaped by previous responses. There are many types of RNNs

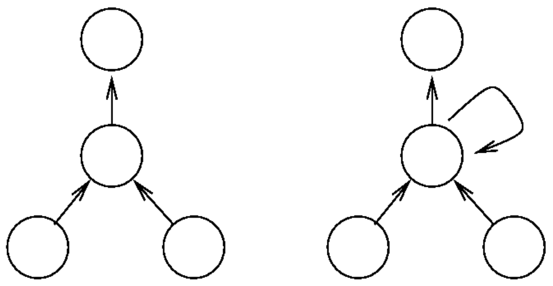

Figure 1-1: A simple feedforward network and a recurrent network with an input layer, one hidden layer containing one processing element, and an output layer.

but they all have two common features. All RNNs make use of some part of the static multilayer perceptron feedforward network and exploit the nonlinear mapping capability of the multilayer feedforward model. The basic distinction between feedforward static networks and RNNs is shown in Figure 1-1. Recurrent neural networks have a feedforward connection for all neurons which allow the network to show dynamic behavior. A network with a fully connected hidden layer, between the input layer and the output layer is shown in Figure 1-2.

Practical constraints often guide the selection of one learning algorithm over another. The learning problem consists of adjusting the parameters (weights) of the network, so that the trajectories have certain specified properties. A common learning algorithm is known as backpropagation. In backpropagation, the weights of the neural network can be adjusted so as to produce an output on the appropriate unit when a particular pattern at the input is observed. The algorithm works by running the training instance through the neural network, and calculating the error between the desired (target) and actual outputs. These differences are then "propagated back" from the output layer to the hidden and input layers in the form of modifications to the weights of each of the component neurons.

1.3 Motivation

Researchers have been trying for half a century to apply human capabilities to machines. Computers are good at performing tasks that humans program them to do. However,

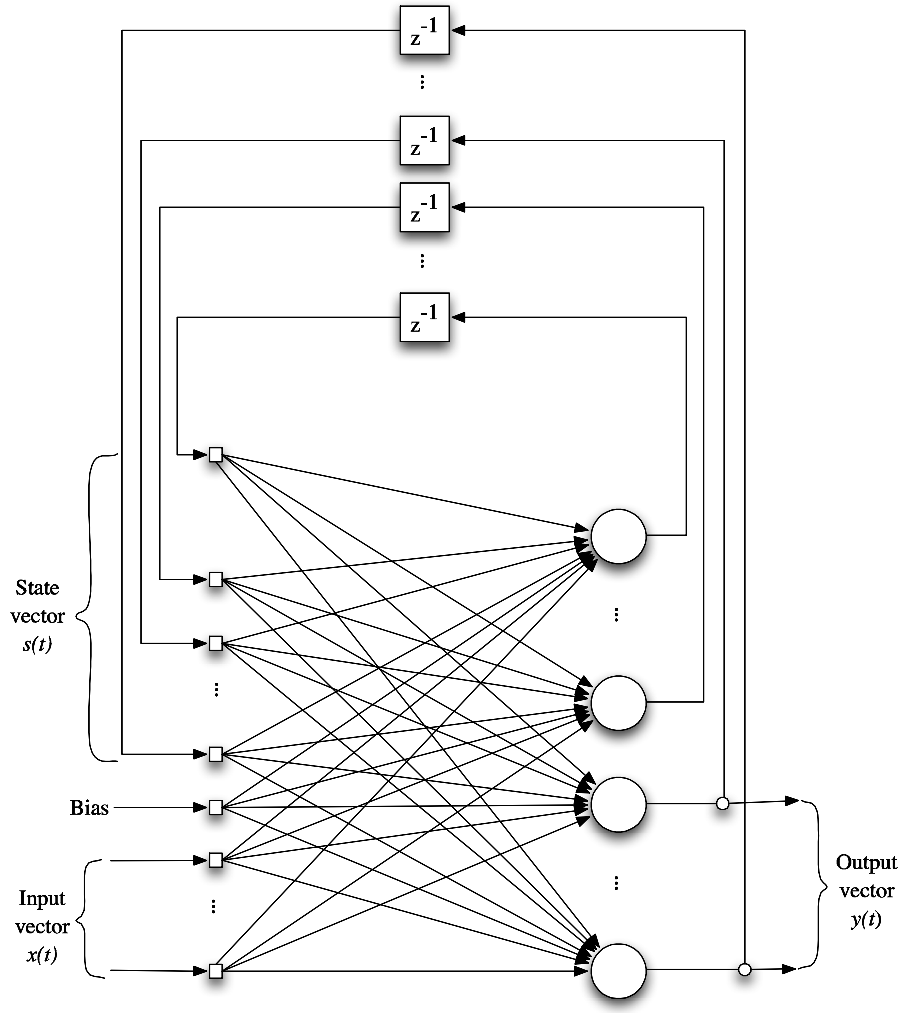

Figure 1-2: A possible architecture for a recurrent neural network.

computers do not perform well when they are tasked with learning a function or a behavior based on prior experience - a skill that humans master very well. In principle, RNNs can implement almost any arbitrary sequential behavior, which makes them promising for adaptive robotics, speech recognition, music composition, attentive vision and many other applications. For comparison, we look at the human brain which itself is a large RNN.

The brain contains on the order of 10^{14} synapses and has a processing speed of 100 Hz [2]. Compared with a modern processor that has 10^8 transistors and a processing speed of 10^9 Hz, the brain does not appear to be much more powerful. The brain, however, is highly parallel and distributed unlike the serial, centralized computation style of the typical processor. Furthermore, the brain is fault tolerant as it has the ability to form new pathways when parts of the brain shutdown. Artificial neural networks, attempt to bring computers a little closer to the brain's capabilities by imitating aspects of information processing in the brain.

RNNs are more complex than perceptron networks because they have feedback loops. This property of RNNs allows them to be better suited for machine learning applications. The biggest drawbacks to RNNs are their enormous computational and storage requirements. For these reasons, it has been impractical to use RNNs in the past. Also, insufficient research has been devoted to designing hardware specifically for RNNs. By designing dedicated hardware, the designer is able to avoid the overhead that comes with a general purpose CPU which results in denser layout of processing elements (PEs).

This thesis focuses on the design, analysis and performance evaluation of a real-time RNN. The RNN algorithm that is presented is based on the RTRL algorithm. In RTRL, each neuron is connected to all other neurons resulting in a fully connected network. The proposed algorithm exploits the fact that synapses are not fully connected in the brain. By disconnecting certain processing elements, we are able to reduce the complexity of the RTRL algorithm while minimizing performance degradation. It focuses on reducing the sensitivities of each neuron to weights associated with its incoming and outgoing connections. The approach results in a localized algorithm that lends itself to hardware realization, while retaining the core capabilities associated with RTRL. The new algorithm is first implemented and simulated using a general purpose CPU and then the design is transcribed to a dedicated hardware platform.

1.4 Thesis Outline

Chapter 2 describes the primary features and limitations of existing RNN algorithms. In order to overcome the inherent computational and storage complexities, we introduce the concepts of ingress and egress sensitivities in Chapter 3. Existing implementations have been largely confined to software realizations, relying on general purpose CPUs, mainly due to the dictated storage and computational requirements. Consequently, this thesis focuses on both software and hardware implementations of the proposed approach.

In Chapter 3 we present the truncated real-time recurrent learning (TRTRL) algorithm. We use simulations to investigate its properties and improvements over the RTRL algorithm. We show that the computational complexity is reduced from $O(N^4)$ to $O(N^2)$ and that storage requirements are reduced from $O(N^3)$ to $O(N^2)$. Furthermore, it is demonstrated that the TRTRL algorithm exhibits minimal performance degradation when compared to the RTRL algorithm while showing vast improvements in speed.

Chapter 4 introduces a hardware architecture realization of the TRTRL algorithm. Due to TRTRL's localized nature, it easily lends itself to a scalable hardware implementation. In addition to the architecture description, we show the dataflow which illustrates the communication protocol between different components of the neural network. We also show simulations and waveforms which demonstrate the abilities, constraints and resource requirements for a FPGA-based hardware realization. Finally, Chapter 5 provides a summary of the contributions made in this thesis, presents future research directions and related publications.

Chapter 2

Literature Review

2.1 Recurrent Neural Network Structures

RNNs, first described by [3], are fundamentally different from feedforward architectures in the sense that they operate not just on an input space but also on an internal state space [4]. Figure 2-1 shows the block diagram of the state-space generic recurrent network. The input layer incorporates a concatenation of feedback nodes and source nodes through which the network is connected to the external environment. The hidden layer neurons define the state of the network. The output of the hidden layer is then fed back to the input layer along a bank of unit delays. The state-space model enables the representation and learning of temporal dependencies over unspecified and potentially infinite intervals according to

$$y(t) = C(s(t)) \tag{2.1}$$

$$s(t) = f(s(t-1), x(t)) \tag{2.2}$$

where C is the matrix of synaptic weights characterizing the output layer and $f(\cdot, \cdot)$ is a nonlinear function characterizing the hidden layer. From this, we see that the the hidden layer is nonlinear, while the output layer is linear. Learning algorithms used for RNNs are usually based on computing the gradient of a cost function with respect to the weights network, as will be described in the next several sections.

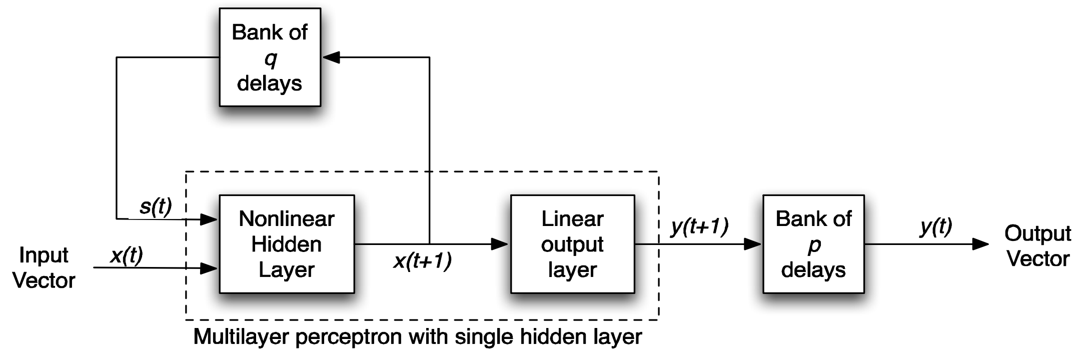

Figure 2-1: The state-space recurrent neural network model.

2.1.1 Elman Network

The simple recurrent network (SRN) described in [5] and depicted in Figure 2-2, has an architecture similar to that of Figure 2-1 except that the output layer may be nonlinear and the bank of unit delays at the output is omitted. The Elman approach calls the bank of unit delays context units, which are also "hidden" because they interact solely with other nodes in the network and not with the outside world. Network processing consists of the following sequence of events. At time t, the input units receive the first input in the sequence. The hidden units feed forward to the output units and at the same time, feed back to the context units. The context units then store the output of the hidden units for one time step, and then feed them back to the input layer. Based on this description, there is only a feedforward cycle, but a learning phase using backpropagation [6] may be used.

By utilizing hidden units and a learning algorithm, the hidden units develop internal representations for the input patterns. These neurons continue to recycle information through the network over multiple time steps, and thereby discover abstract representations of time. Therefore, we say that the context units provide the network with dynamic memory so as to encode the information contained in the input pattern and remember the previous internal state. In the next section, we will discuss specific learning algorithms along with their application to recurrent neural networks.

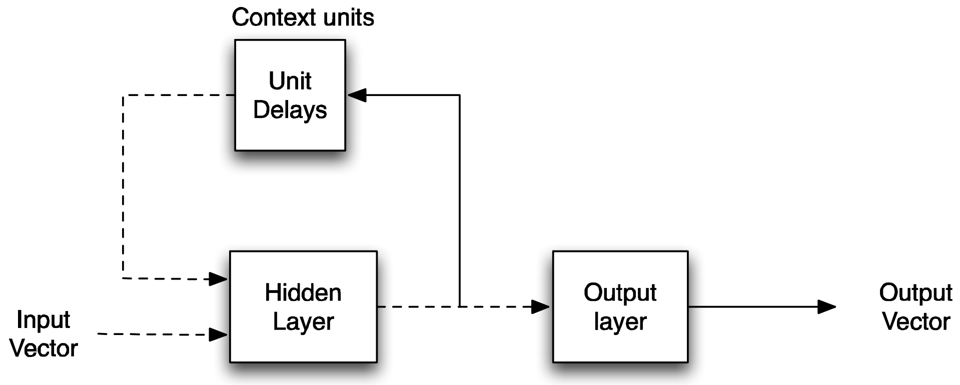

Figure 2-2: The Elman simple recurrent network where activations are copied from the hidden layer to the context layer and then fed back into the hidden layer after a one time step delay. The dotted lines represent trainable connections.

2.2 Learning Algorithms

There are two modes of training static feedforward neural networks: batch mode and on-line mode. In batch mode, the sensitivity of the network is computed for the entire training set prior to adjusting the parameters of the network. In the online learning mode, conversely, adjustments are made after each input pattern is presented in the training set. Similarly, there are two modes of training recurrent neural networks: epochwise and continual operations [7].

2.2.1 Epochwise Training

In epochwise training, for a given epoch, the recurrent network runs from a particular initial state until it reaches a stopping state. The network could stop training, for example, after a length of time has passed. Once the training is stopped, the network is reset to an initial state for the next epoch. The initial state is not required to be the same for each new epoch of training. What is important, however, is that the initial state for the new epoch is different from the state at the end of the previous epoch. In the current terminology, the epoch in the recurrent network corresponds to one training pattern for ordinary static feedforward networks.

2.2.2 Continuous Training

In contrast to epochwise training, situations arise when no reset states are available and on-line learning is required. The distinguishing feature of continuous training is that the network learns at the same time as it operates. An example for which an on-line learning process is required is in modeling of a non-stationary process such as a speech signal. In this case, the continual operation of the network does not offer a convenient time at which to stop the training and begin a new epoch with different values for the free parameters of the network. Due to the importance of on-line learning, this thesis will focus on learning algorithms that facilitate continuous training, as described in the next section.

2.3 Gradient-Based Network Training

We next extend our discussion to training recurrent networks using the two types of algorithms described in Section 2.2. We will describe two different learning algorithms for recurrent neural networks: backpropagation through time and real-time recurrent learning. These algorithms are both based on the method of gradient descent, where the instantaneous value of a cost function is minimized with respect to the synaptic weights of the network.

2.3.1 Backpropagation Through Time

The backpropagation through time (BPTT) algorithm can be viewed as an extension to the classical Elman network described in Section 2.1.1 and is a generalization of backpropagation for static networks. Various batch-training forms of the algorithm have been derived by [8] who later published his approach in [9]. Other versions were derived and discussed in [6] and [10]. The algorithm is depicted in Figure 2-3.

Let us denote $\mathcal{N}$ as a recurrent network require to learn a temporal task, starting at time t_0 and ending at time t. Next, let us denote $\mathcal{N}^*$ as the feedforward network that results from unrolling the temporal operation of recurrent network $\mathcal{N}$, where $\mathcal{N}^*$ has a layer for each time step in the time interval $[t_0, t]$ and n units in each layer. Each neuron in the network $\mathcal{N}$, there is a copy of each layer $\mathcal{N}^*$. Each connection from unit i to unit j in $\mathcal{N}$

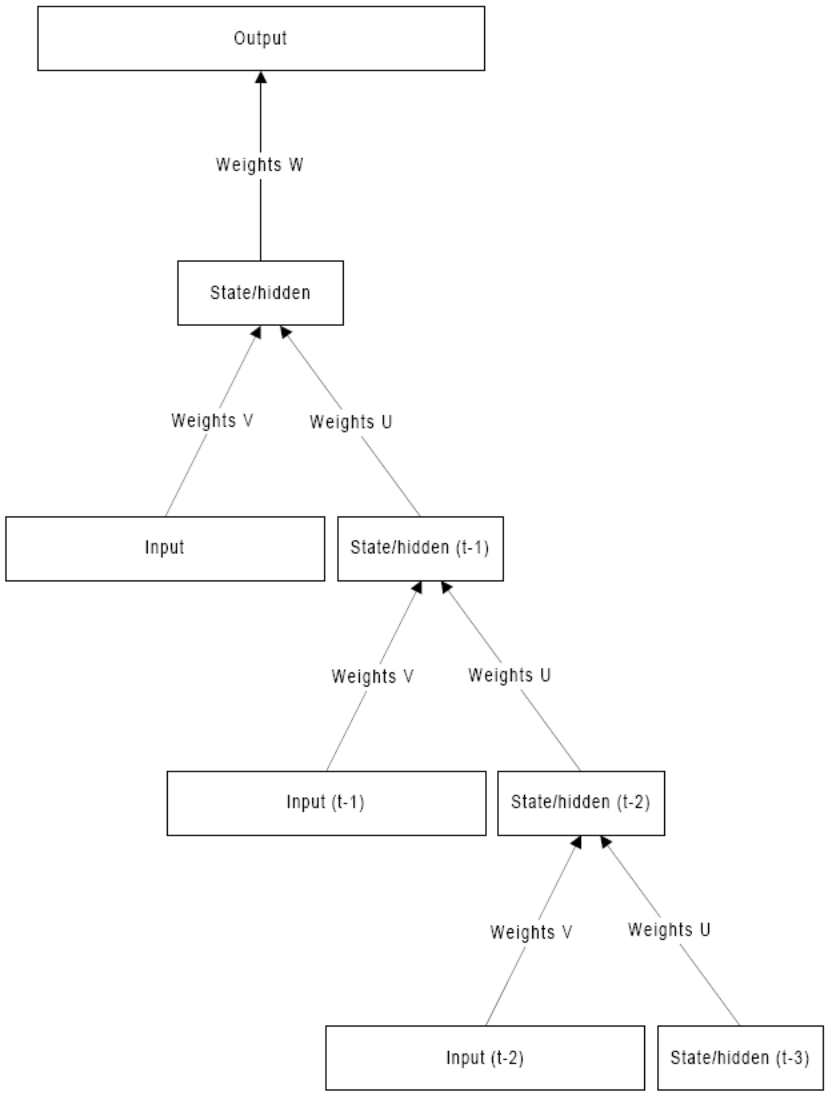

Figure 2-3: Unfolding of the BPTT algorithm for $l = 3$ [1].

has a copy connecting unit j in layer l to unit i in layer $l+1$, for each time step $l \in [t_0, t]$.

In the first phase, a copy of the whole RNN is added to the top of a growing feedforward network on each update cycle which updates the internal states of the network. Thus, if the network is to process a signal that is t time steps long, then copies of the network are created and the feedback connections are modified so that they are feedforward connections from one network to the subsequent network. Second, backpropagation is used to update the weights with respect to the performance error. In the second phase, the network is trained using backpropagation to update the weights with respect to the performance error. It becomes one large feedforward network with the updated weights being treated as shared weights.

The key advantage of BPTT is that the training algorithm, backpropagation, is identical to those that are used for feedforward networks and therefore it can be applied to a wide variety of problems. However, the epochwise BPTT algorithm [11] has several fundamental drawbacks: first, it is not a real-time algorithm in the sense that batch data must be applied and second, the algorithm has extensive memory requirements that are dictated by the need to store growing amounts of state information. The procedure works well for relatively simple recurrent networks consisting of a few neurons as it has a computational complexity of $O(N^2)$, however the memory requirements of the underlying formulas become too large when the procedure is applied to more general architectures that are typical of those encountered in practice.

Other, continuous time approaches to training recurrent networks to handle time-varying input or output have been investigated by [12]. Unfortunately, this approach uses a restrictive architecture that is not suitable for more complex problems.

2.3.2 Real-time Recurrent Learning (RTRL)

First outlined in [13], the real time recurrent learning (RTRL) algorithm uses the same network structure as depicted in Figure 1-2. It is important to note, that the RTRL structure differs from the Elman network as there is no distinct output layer present and any node in the network is allowed have a target value. RTRL and its variants calculate the gradients in real-time making these algorithms very attractive in that they are applicable to applications where data is continuosly presented to the network. In order to allow real-time training of patterns of indefinite duration, it is useful to make the weight changes while the network is running. This is an important feature because there is no need to set epoch boundaries for training the network: the network can train without end. This leads both to a conceptual and implementational simplification of the procedure. The gradients at time t are obtained in terms of those at time instant $t-1$. Once the gradients are evaluated, weight updates can be calculated in a straightforward manner. This procedure of updating each weight, w_{ij}, is analogous to the commonly used method of training a feedforward network. Weight changes are performed after each pattern is presented rather than accumulating them elsewhere and them making the net change following each complete pattern cycle.

Algorithm Description

In this section we provide a detailed description of the RTRL algorithm. Let us assume that a network consists of a set of N fully connected neurons and a set of M inputs. Further, $K \in N$ will denote the set of neurons for which there is a target. Let $w_{ij}(t)$ denote the weight (i.e. the synaptic strength) associated with the link originating from neuron j towards neuron i at time t. The net input to neuron k, $s_k(t)$, is defined as the weighted sum of all activations in the network, $z_l(t)$. Based on standard RTRL terminology, we define the activation function of node k at time $t+1$ to be

$$y_k(t+1) = f_k\left(s_k\left(t\right)\right), \tag{2.3}$$

where

$$s_k(t) = \sum_{l \in N \cup M} w_{kl} z_l(t), \tag{2.4}$$

$$z_k(t) = \begin{cases} x_k(t) & \text{if } k \in M \\ y_k(t) & \text{if } k \in N \end{cases} \tag{2.5}$$

and the non-linear activation function, $f(\cdot)$, maps $s_k(t)$ to the range [0,1]. The overall network error at time t is defined by

$$J(t) = \frac{1}{2}\sum_{k\in T}\left[d_k(t) - y_k(t)\right]^2 \tag{2.6}$$

$$= \frac{1}{2}\sum_{k\in T}\left[e_k(t)\right]^2 \tag{2.7}$$

where $d_k(t)$ denotes the desired target value for output k at time t. Correspondingly, the error is minimized along a negative multiple of the performance measure gradient such that

$$w_{ij}(t+1) = w_{ij}(t+1) + \alpha\frac{\partial J(t+1)}{\partial w_{ij}(t)}. \tag{2.8}$$

The online calculation of the gradients is achieved by exploiting the following relationship:

$$-\frac{\partial J(t)}{\partial w_{ij}(t)} = \sum_{k\in T} e_k(t)\frac{\partial y_k(t)}{\partial w_{ij}}. \tag{2.9}$$

By identifying the partial derivatives of the activation functions with respect to the weights as sensitivity elements, and denoting the notation by

$$p^k_{ij}(t) = \frac{\partial y_k(t)}{\partial w_{ij}}, \tag{2.10}$$

we obtain the following recursive equation:

$$p^k_{ij}(t+1) = f'_k\left(s_k\left(t\right)\right)\left[\sum_{l\in N} w_{kl}p^l_{ij}(t) + \delta_{ik}z_j(t)\right], \tag{2.11}$$

where $p^k_{ij}(0) = 0$ and δ_{ik} is the Kronecker delta. Equations (2.11) and (2.9) allow one to obtain the performance gradient at any given time.

Algorithm Complexity

As can be seen from eq. (2.11), each neuron is required to perform $O(N^3)$ multiplications yielding an overall complexity of $O(N^4)$. Moreover, the storage requirements are dominated by the weights $O(N^2)$ and, more importantly, the sensitivity matrices, $p^k_{ij}(t)$, which are $O(N^3)$. Due to the distributed nature of the network, the calculation can be reduced to significantly by having each neuron compute its sensitivities in parallel. If performed in hardware, these computation processes can be accelerated by exploiting pipelining and module replication. However, unlike the computational requirements, the storage requirements cannot be reduced as they constitute a crucial component in the weight update procedure.

2.3.3 RTRL Improvements

Several schemes that are presented in this section aim to reduce the storage complexity associated with RTRL. A unifying theme of these methods comprises of subgrouping the neurons into multiple, non-overlapping subnetworks. Although the computational gain is significant, the storage requirements remain high, in particular when a small set of subgroups is employed. Hybrid BPTT/RTRL schemes have received attention in [14], reducing the complexity to $O(N^3)$. This technique takes blocks of BPTT and uses blocks of RTRL to encapsulate the history before the start of each block. This method, however, still relies on the batch learning of BPTT. Other methods, that similarly reduce computational complexity to $O(N^3)$, have proposed using Green's function [15].

Subgrouping

In [16], the sensitivity set for each neuron is reduced to a subgroup of neurons, thereby decomposing the network into several non-overlapping sub-networks. This algorithm reduced the complexity of RTRL by simply leaving out elements of the sensitivity matrix (eq. (2.11)) based upon a subgrouping of the nodes. The key advantage of subgrouping in this manner is the immediate reduction in computations to $O(N^4/g^3)$, where g denotes the number of groups. However, for a small number of subgroups, the advantage becomes negligible. If g is large, there is little crossover of training information from different groups, thereby significantly reducing the network's capabilities. The arbitrary selection of subgroups also appears somewhat weak. To address this concern, recent work has suggested dynamically partitioning the groups in accordance with gradient information being calculated online [17]. Although it constitutes a more intelligent and data-dependent approach, the method is not scalable due to the complex process of dynamically redefining the subgroup boundaries.

Moreover, the key problems associated with the number of groups created in [16] remain.

Stochastic Algorithms

Gradient based method require the knowledge of the network architecture and the computation of the activation function's derivative for each neuron. Stochastic based methods do not have such constraints and are able to find the global minimum of an error function. A stochastic learning algorithm based on simulated annealing (SA) techniques was proposed in [18] and showed that the chance of the algorithm being stuck in a local minimum is much smaller than that of gradient based methods. A genetic algorithm (GA) based method was introduced in [19]. The underlying principle in GA is evolution theory and the fact that nature has no memory and knowledge about the environment. These assumptions enable GA to be appropriate for function approximation. The experiments performed in [19] dealt with finding chromosomal population patterns and the GA method successfully found the global optimum while gradient descent methods were trapped in local minimums frequently.

RTRL was compared with simulated annealing and genetic algorithms in [20] using the Henon [21] deterministic chaos model, where it was determined that RTRL was able to attain the lowest mean squared error yet it required a much longer training time. The GAs performed better than gradient based RTRL when the training patterns were repeatedly presented to the networks and for network problems where outputs need not be accurately matched to the desired outputs.

2.4 Hardware Considerations

In order to optimize the hardware implementation of the RNN algorithm, several considerations must be taken into account. Because the RTRL algorithm inherently needs many interconnects and local storage, it is difficult to realize this learning method as a very large-scale integrated system (VLSI). The learning method proposed in this thesis is more localized and resource efficient which will be shown to yield a scalable VLSI implementation. Several key components of the neural network are considered next.

2.4.1 Nonlinear Activation Function

The nonlinear activation, or squashing, function used in the RTRL algorithm is a sigmoid

$$f(x) = \frac{1}{1 + e^{-x}} \tag{2.12}$$

the output of which is constrained between $(0, 1)$. The derivative of the sigmoid can be shown to satisfy the following equation:

$$\frac{df(x)}{dx} = f(x)(1 - f(x)). \tag{2.13}$$

In a VLSI realization of large-scale RNNs, where resources are heavily limited, the best approach for implementing the activation function is by means of piece-wise linear (PWL) implementation of the sigmoid and its derivatives. A comprehensive method described in [22], suggests using powers of 2 to construct the linear segments. This turns into a very convenient method because it consists only of shifts and adds. This approach proposes a 15-segment PWL with integer intervals $[-7, 8]$. If the input value $x > 0$, the segments are defined by the formula $1 - 2^{-n}$ where n is the discrete interval on which the input lies. If the input value $x < 0$, the segments are defined by the formula 2^{-n}. The design provides a close approximation of the sigmoid function and furthermore saves on chip resources.

2.4.2 Learning Rate

In order to save resources and improve performance, we will use a learning rate parameter α that will be a power of 2. When it is necessary to update the weights, the value of Δw_{ij} will simply be shifted in order to simulate multiplication by a power of 2.

2.4.3 Data Representation

Precision

The nature of the neural networks requires precise calculations. For this reason it is important to consider the implications of fixed or floating point notation. Floating point architectures are bulky and become impractical for an application with high resource constraints such as a RNN. Therefore we choose a fixed point notation. It has been determined in [23] that it is possible to use integer weights for some applications, however, 18-bit fixed point precision is sufficient to represent the necessary data in a larger set of applications. For data representation in our hardware implementation, we will use 20-bit fixed point numbers.

Multiplier Structure

Multiplication is a necessity in neural networks however it is a complex task for FPGAs. Our goal is to use the lowest cost (lowest storage requirements) multipliers in the hardware implementation. A Booth multiplier [24] only needs 1 register and l clock cycles to compute the product, where l denotes the length of the multiplicand. A problem also arises in determining the location of the mantissa in the product of 2 numbers. In order to solve this problem, recently, the VHDL working group (part of IEEE) has been working on a synthesizable fixed point multiplication library [25]. The library provides functions and datatypes for easy manipulation of fixed point numbers.

2.5 Performance Evaluation Metrics

In order to test and compare RNNs, we must choose tasks that exploit their proficiencies. The algorithms were tested on tasks that require the network to learn to configure itself so that it stores information computed from the input stream at earlier times in order to determine the output at later times. In other words, the network must to learn to configure itself in such a way that temporal dependencies are required. In this way, the network learns to represent useful state information internally so as to successfully accomplish these tasks.

2.5.1 Frequency Doubler

A very basic task for an RNN to perform is the frequency doubler [13]. Depicted in Figure 2-4, the frequency doubler requires the network to recognize that two inputs having the same value, correspond with different outputs. For example, the value of $\sin(\frac{\pi}{4})$ is equivalent to $\sin(\frac{3\pi}{4})$, yet these inputs correspond to different values if the period is doubled. To that

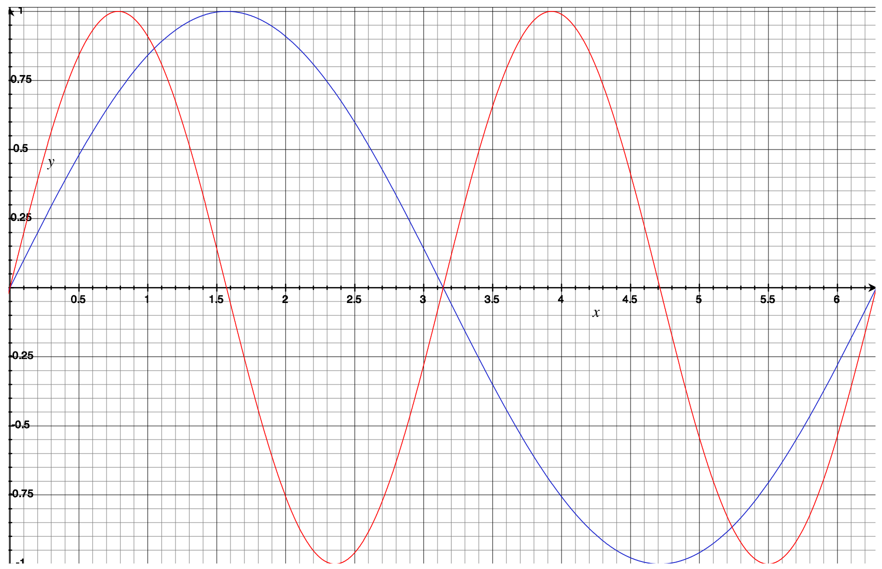

Figure 2-4: The input (blue) and target output (red) curves for the frequency doubler simulation task.

end, the frequency doubler is basic test for memory, or state inference.

2.5.2 Chaotic Time Series Prediction

Since the introduction of artificial neural networks, they have been extensively used for the task of time series prediction [26]. The latter is of particular interest in the context of RNNs because these networks are able to capture temporal dependencies thereby improving the overall prediction capability. The problem of time series prediction can be described as follows: Given some history of values of a particular entity $x_1,\ x_2, ...,\ x_t$ observed at discrete time instances t, find the value of the entity at time instant $t+\tau$ [27]. The time series chosen required the networks to predict future values of the Mackey-Glass (MG) chaotic series [28], which has been extensively used as an RNN benchmark [29].

The MG series, first proposed as a model for the production of white blood cells, is based on the time-delayed differential equations with adjustable parameter τ,

$$\frac{\partial x(t)}{\partial t} = \frac{ax(t-\tau)}{1+x^c(t-\tau)} - bx(t). \tag{2.14}$$

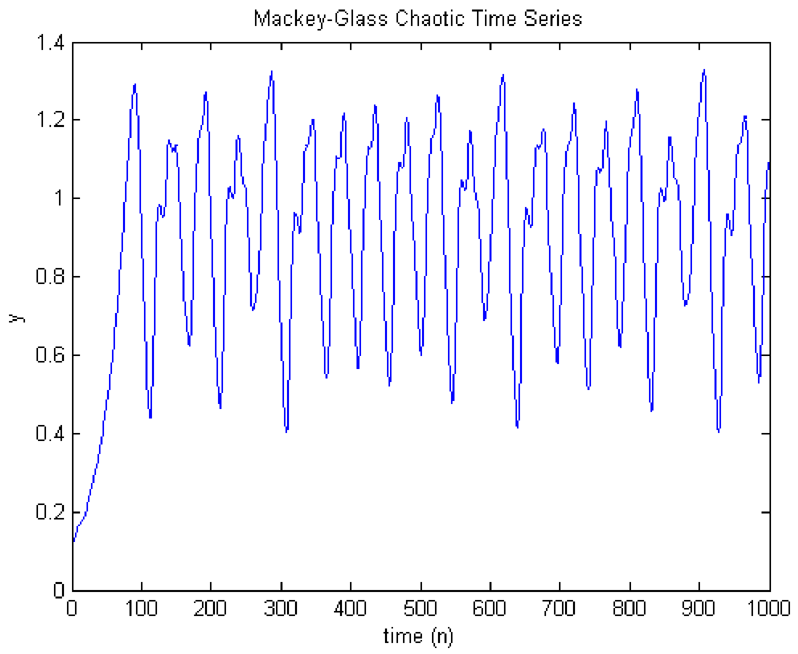

Figure 2-5: The input data for the Mackey-Glass chaotic time series prediction task.

To obtain values at integer time points, the fourth-order Runge-Kutta integration scheme was used as means of finding the numerical solution to the above MG equation [30]. The plot of the first 1000 points of the MG series using $a = 0.2$, $b = 0.1$ and $c = 10$ is shown in Figure 2-5. These parameters were used when performing simulations on network algorithms.

2.5.3 Sequence Memorization

Another fundamental testbench that appears often in the literature is sequence memorization, which examines the capabilities of the network to generate sequence values based on temporal dependencies. In its basic form, the inputs to the network are either *high* or *low*. After the network receives a *high* or *low* signal, following n time steps, it must output that same respective signal. A sample input - target output sequence is shown in Table 2.1.

Based on the inputs, the neural network must determine how many time steps it must wait before it outputs the original signal. In this manner, the network must store, or memorize, the input signal within its nodes for a certain duration of time.

Table 2.1: A sample input sequence and the corresponding target outputs with a memorization 3 time steps

Input	**Target**
-1	0
0	0
0	0
0	-1
1	0
0	0
0	0
0	1
1	0
0	0

2.5.4 Error Measurement

In all of the studied simulation tasks, performance was measured and compared by computing the mean-squared-error (MSE) at the end of each epoch, which is defined as one input test pattern, at the end of which a weight update is performed. The MSE at time τ is given by

$$MSE(\tau) = E[(T(\tau) - y(\tau))^2] \tag{2.15}$$

where $T(\tau)$ denotes the target value and $y(\tau)$ the actual output of the neural network.

Chapter 3

Truncated Real-Time Recurrent Learning (TRTRL)

3.1 Algorithm Description

TRTRL is a variation on RTRL that is specifically designed to overcome the scalability limitations of RTRL while retaining its key performance attributes. TRTRL accomplishes this goal by reducing the amount of information that each neuron is required to consider as it performs its computations. Let us begin with several key definitions that will help guide us through the discussion. Figure 3-1 is provided for reference.

Definition 1 *Let I_j denote the set of nodes that have a direct link (and, hence, a unique associated weight) to node j. We shall refer to this set as the ingress set of node j.*

Definition 2 *Let E_j denote the set of nodes that node j has a link (and, hence, a unique associated weight) to. We shall refer to this set as the egress set of node j.*

It should be noted that a node can reside within both ingress and egress sets of another node. Moreover, for the purpose of notation convenience, we shall consider the feedback (i.e. recurrent) link that each node has to itself, to be part of the node's egress set. Consequently, the basic assumption in TRTRL is that the sensitivities of each neuron are limited to its ingress and egress set. This means that, coarsely speaking, a neuron's activation is not

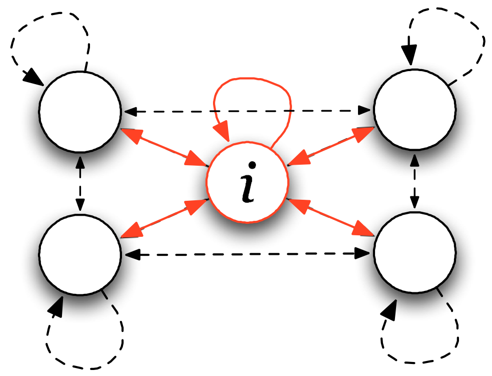

Figure 3-1: Comparison of the sensitivity set of neuron i. The black and red lines represent sensitivities considered in RTRL, while the red lines represent those present in TRTRL.

directly sensitive to any weight that is not in the neurons ingress or egress set. The only exception to this rule pertains to neurons with targets, as will be elaborated on later.

By localizing the information processed by each neuron, the calculation of eq. (2.11) comprises of three main components. First, its ingress sensitivity function is given by

$$p_{ij}^{i}(t+1) = f_i'\left(s_i\left(t\right)\right)\left[w_{ij}p_{ij}^{j}(t) + z_j(t)\right] \;\; \forall \; i \notin K, i \neq j \tag{3.1}$$

where i and j are nodes of the network and $K \in N$ denotes the set of neurons for which there is a target. Notice that the summation from eq. (2.11) is reduced to a single multiplication since $p_{ij}^{l} = 0$ for all $l \neq i$. Secondly, following a similar rationale to that of the ingress set, the sensitivities pertaining to the egress set of node i are given by

$$p_{ji}^{i}(t+1) = f_i'\left(s_i\left(t\right)\right)\left[w_{ij}p_{ji}^{j}(t) + \delta_{ji}y_i(t)\right] \;\; \forall i \notin K. \tag{3.2}$$

From the above two expressions it becomes evident that the aggregate computational load for each neuron is $O(N)$ (in fact, rather close to $2N$).

In order to complete the description of TRTRL, an update rule must be derived for the output neurons, for which we once again refer to eqs. (2.8) and (2.9). Here, there are two possible scenarios in the first of which the number of output layer neurons is significantly smaller than the total number of neurons in the network: $K \ll N$. In this case, it is expected that the majority of the information will be represented by weights and signals associated with the non-output neurons, in which we make the assumption that the output neurons do not have connecting weights, i.e. $w(i,j) = 0 \; \forall \; i, j \in K$. For the output neurons, a non-zero sensitivity element must exist in order to provide gradient information required by the weight update rule, as shown in eq. (2.8). To comply with this requirement, a direct link is added from each output neuron to each of the N neurons in the network. Consequently, each output neuron, $o \in K$, performs a sensitivity update for each weight in the network. This is can be achieved using the following update rule:

$$p_{ij}^{o}(t+1) = \begin{cases} f_o'\left(s_o\left(t\right)\right)\left[w_{oi}p_{ij}^{i}(t) + w_{oj}p_{ij}^{j}(t) + \delta_{io}z_j(t)\right] \\ \text{if } i \neq o, \;\; i \notin K, \text{ and} \\ 0 \;\; \text{otherwise} \end{cases} . \tag{3.3}$$

The advantages of following such an update rule are that computations are kept to a minimum, while high information content is retained due to the structure of the network.

Alternatively, for networks in which the number of neurons with targets is almost the same as the number of neurons in the network $K \approx N$, eq. (3.3) suggests that very few of the weights will be non-zero. This gives rise to the need for a revised formulation of eq. (3.3). If the recurrent weights of output neurons are non-zero, the respective update rule becomes:

$$p_{ij}^{o}(t+1) = \begin{cases} f_o'\left(s_o\left(t\right)\right)\left[w_{oi}p_{ij}^{i}(t) + w_{oj}p_{ij}^{j}(t) + \sum\limits_{l \in T} w_{ol}p_{ij}^{l}(t) + \delta_{io}z_j(t)\right] \\ \text{if } i \neq o \text{ and } j \neq o, \text{ and} \\ f_o'\left(s_o\left(t\right)\right)\left[\sum\limits_{l \in T} w_{ol}p_{ij}^{l}(t) + \delta_{io}z_j(t)\right] \;\; \text{otherwise} \end{cases} . \tag{3.4}$$

The partial sensitivity matrix is invariant to the fact that there may still exist a unique weight between any two neurons in the network. The full calculation of eq. (2.11) must be performed for all input, bias and output units. The only difference between TRTRL and RTRL, in this context, is that neurons are limited in the sensitivities.

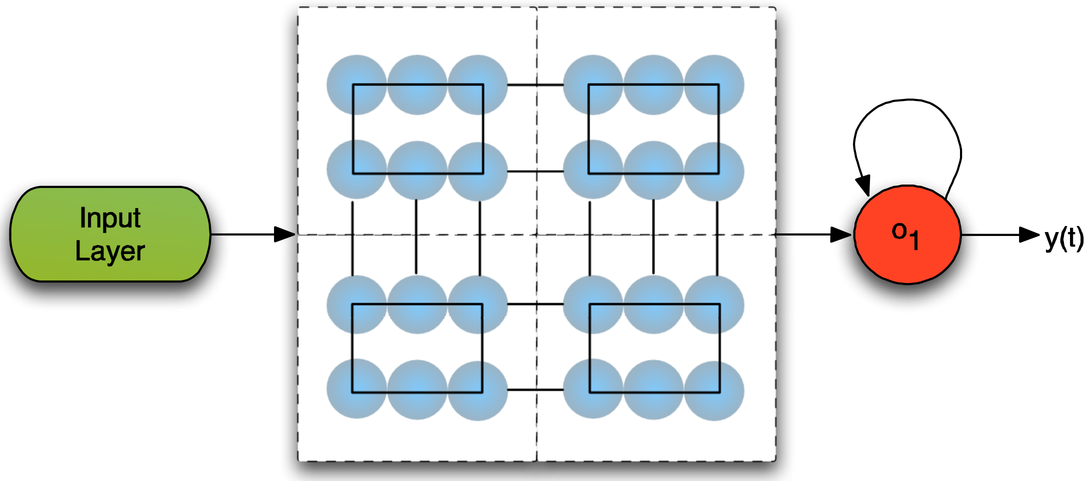

Figure 3-2: A diagram of clustered TRTRL with 4 clusters of 6 neurons each. Neurons compute the sensitivities to weights that are within that cluster or that are connected to another cluster via an external link.

3.1.1 Clustering

In order to improve the TRTRL algorithm for hardware scalability, the number of connections between neurons was further reduced. Following a similar approach first discussed in [16], clusters of neurons are formed where the neurons in each cluster are only connected to and therefore sensitive to neurons within the same cluster. This implies that each hidden layer node i only sees N/B other nodes, where B is the number of clusters. The rest of the architecture in this approach remains the same where all nodes receive the input patterns from the input layer and all nodes are connected to the output layer. It is noted that there must be some connectivity between clusters, defined as inter-cluster connectivity, thus some neurons within each cluster are connected to other neurons in other clusters by means of a direct link. A diagram of the clustered layout is shown in Figure 3-2. By clustering, we further reduce the number of connections of each processing element.

Clustering Storage Complexity

In a partially connected TRTRL network, the exact storage requirements are $(2B + M + B + 1)(N - K)$ for all hidden layer neurons and $(N + M + 2N + 1)K$ for all K_i output layer neurons. Clustering neurons is useful when implementing the network in hardware because of the decreased number of connections between nodes. This also leads to better memory management. Both of these factors allow for more neurons to be placed on a FPGA chip. Clustering TRTRL will be further described in Chapter 4.

3.1.2 Storage and Computational Complexity

The primary benefits of TRTRL, from an implementation perspective, are the substantial reductions in computation complexity and storage requirements. Computation time is dominated by the calculation of the sensitivity elements. While in the original RTRL scheme, each neuron was required to perform $O(N^3)$ operations, TRTRL requires only $O(N)$.

A similar reduction in resources is observed in the storage requirements of TRTRL. In RTRL, each node requires sensitivity N^2 elements, while TRTRL only needs $2N$, $M + N$ weights and 1 activation per neuron. As such, the overall storage requirement drops from $O(N^3)$ to $O(N^2)$. It should also be noted that, as opposed to RTRL, TRTRL is a highly localized algorithm as nodes are no longer required to communicate with other neurons that may be located at a distant part of the network. This contributes to the more effective implementation prospect of the scheme in hardware. Moreover, it is interesting to note that the TRTRL formalism is not restricted to fully connected networks as was detailed in Section 3.1.1. In fact, assuming that each node is only connected to M other nodes, the computational complexity becomes $O(KMN)$ while storage is reduced to $O(MN)$. The only constraint imposed in such cases is that each node have a direct link to the output neurons (as means of propagating error information), as dictated by eq. (2.9).

Memory Requirements Example

Let us provide an example of the actual differences between RTRL and TRTRL in constructing the largest possible network using an existing FPGA device. For the non-clustered version from the discussion above, each neuron is connected to N other neurons (i.e. there is a weight between neuron i and N other neurons). The network has N hidden layer processing elements and M inputs. Table 3.1 summarizes the storage requirements per neuron of the two methods. We will assume we are using one of the latest Altera Stratix II FPGA

Table 3.1: A comparison of the storage complexity of RTRL and TRTRL

	RTRL	**TRTRL**
Sensitivities to other neurons	N^2	$2N$
Weights	$M+N$	$M+N$
Activations	1	1
Maximum Possible Number of Neurons	50	204

devices, which has up to 2.5 Mbit of integrated block memory and that each weight and sensitivity value will be represented by using 20-bit fixed point precision numbers.

We can conclude that by employing TRTRL, a maximum 308% increase in neuron density can be achieved on this platform. This increase is theoretical at best because our analysis does not specifically compare computational complexity as it is highly specific to the actual network design in hardware. We can see however, that RTRL requires $O(N^3)$ calculations per neuron. Combining this observation and the fact that RTRL requires many more bus and interconnect lines than the more localized TRTRL algorithm, the maximum possible number of neurons is reduced by an even greater amount.

3.2 Simulation Results

We performed a comparison between the RTRL and TRTRL algorithms on several commonly employed testbenches described in Section 2.5. These tasks required the network to learn to configure itself in such a way that temporal dependencies are required, i.e. information that arrives at a given time has strong impact on the value of outputs at later times. In other words, the network is required to learn to represent useful state information internally so as to successfully accomplish these tasks. We will first discuss factors that affect the simulations.

3.2.1 Simulation Considerations

Because of the nature of RTRL and TRTRL, there are a number of initial parameters that can be modified in order to achieve the varied results, the most important of these being the initial weights. Several options can be applied when initializing the weights, however it was found through simulation, that the best way to ensure convergence of the network is to initialize the weights to be within the upper and lower bounds of target output range.

Momentum

Secondly, there are several schemes that were tested in setting network learning parameter, α which is in the range of $[0,1]$. If the learning parameter is set too large, the weights may escalate to infinity and the activations will grow out of control. If the learning parameter is set too small, the network will take too long to learn or may be stuck in a local minimum and never learn at all, again denying the network from convergence. The basic method of modifying the learning rate is by trial and error, which unfortunately is impractical. One method of assisting the learning rate is to include a momentum term in the weight update calculation [31]. With momentum term m, the weight update rule becomes

$$w_{ij}(t+1) = w_{ij}(t+1) + \alpha(m\Delta w_{ij}(t-1) + \Delta w_{ij}) \tag{3.5}$$

where m, also determined by trial and error, is $0 < m < 1$ but is usually much closer to 0 than 1. The goal of the momentum term is to steer the weight changes based upon previous weight changes. When the gradient changes direction often, momentum will smooth out the variations. This is particularly useful when the network is not well-conditioned.

Modifying the Learning Rate

Another method is to calculate the learning rate dynamically using the adaptive learning rate algorithm discussed in [32]. This modifies the learning rate at the end of each time step t before the weight update isp erformed. The method is given by the following:

$$\alpha_t = \frac{\langle \delta(t-1), \delta(t-1) \rangle}{\langle \delta(t-1), \psi(t-1) \rangle}, \tag{3.6}$$

where $\delta(t) = w(t) - w(t-1)$, and $\psi(t-1) = \Delta w_{ij}(t) - \Delta w_{ij}(t-1)$ and $\langle \cdot, \cdot \rangle$ denotes the standard inner product. This method requires storing the 2 previous weight matrices as well as the 2 previous error gradients. This is a feasible method to use in software as it only doubles the storage requirements of the weights, but is much more costly in a hardware implementations. We note that this scheme could be extended to adaptively calculate a respective learning rate α_{ij} for each weight w_{ij}, but this method introduces complexity to the problem. The adaptive learning rate could also be combined with the momentum term described in section 3.2.1 in order to impose greater control on the gradients.

3.2.2 Frequency Doubler

The first task chosen was the frequency doubler system. For this task, the networks were required to produce a sinusoidal signal that has twice the frequency of the signal applied at its input. The latter is a sinusoid with a 16-sample period while the desired output signal is a sinusoid with an 8-sample period. This is a suitable basic task for the network as the input to output mapping is a nonlinear function. For both RTRL and TRTRL, the network consisted of a single hidden layer with 15 fully recurrent neurons, one bias input neuron whose (constant) value is 1 and a single linear output neuron. The same set of random initial weights was applied to both networks each time they were trained.

Figure 3-3 depicts the average learning curves for both algorithms over 10 runs. During the first 200 epochs, both algorithms train with the same rate but afterwards, the RTRL algorithm improves faster than TRTRL. The difference in the output accuracies becomes miniscule with time, however. This basic test case illustrates the inherent sufficiency of the gradient calculations in TRTRL.

The simulation was performed several times with varying numbers of hidden layer neurons, all simulations achieved similar results to Figure 3-3. Simulations were also performed on a clustered version of TRTRL. The clustered version of TRTRL used in this simulation is not a fully connected recurrent network (see Figure 1-2). Employing a network structure similar to the one shown in Figure 3-2, except using 5 clusters of 3 neurons each for a total of 15 neurons, the network was able to accurately predict the frequency doubler sequence. In this experiment, as can be seen in Figure 3-4 and compared to Figure 3-3, the simulation reached an acceptable error of 10^{-3} at the same time as the fully-connected TRTRL.

3.2.3 Chaotic Time Series Prediction

The next task performed was chaotic time series prediction, that was described in section 2.5. Please refer to eq. (2.14) and Figure 2-5 for reference. For this simulation, we assume

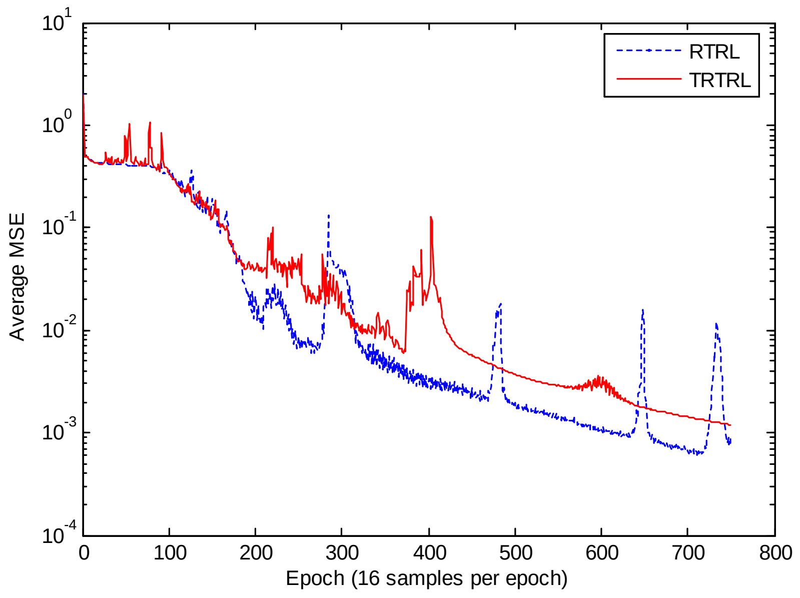

Figure 3-3: Average learning curves for the frequency doubler simulation showing RTRL and TRTRL.

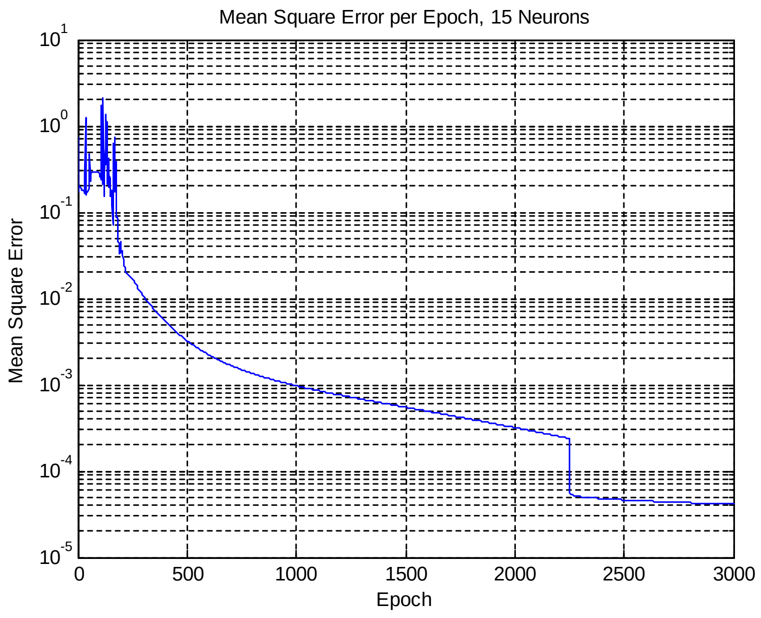

Figure 3-4: A network of 15 neurons, with clusters of 3 neurons performing the frequency doubler task.

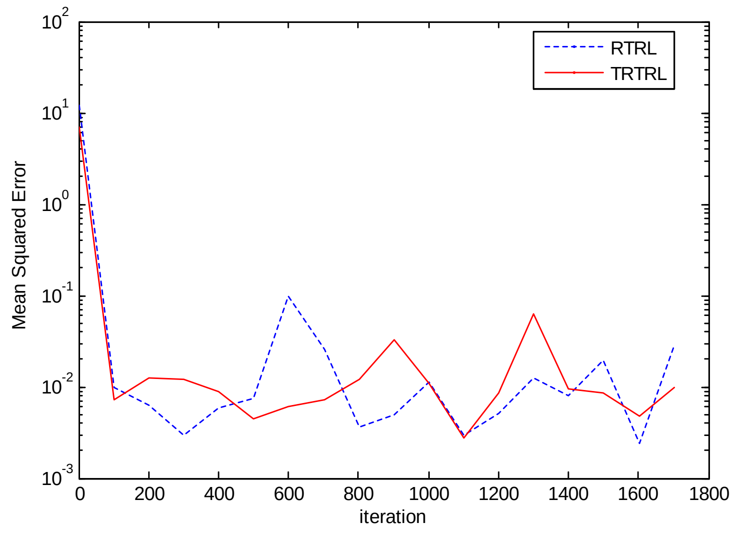

Figure 3-5: Learning curves for the chaotic time series prediction task, applied to both RTRL and TRTRL.

that the time step is 1, $x(0) = 0.1$, $t = 17$ and $x(t) = 0$ for $t < 0$. The task is to predict the value of $x(t+30)$ given the current input and internal state representation. The chaotic time series prediction task was chosen because it is significantly more difficult for the networks to solve than the frequency doubler. The network topology was identical to the one used for the frequency doubler task. Both networks were constructed using 25 fully recurrent neurons, one bias input neuron whose (constant) value is 1 and a single linear output neuron. The same set of random initial weights was used for each network for each training run in order to maintain consistency between the algorithms.

Figure 3-5 illustrates the learning curves for both algorithms. After completing 1800 time steps of training, both networks learned to predict the output 30 time steps ahead of the input with a similar degree of accuracy. This simulation task proved difficult for both networks as the error did not drop below 10^{-3}. Once again, however, the outcome of this simulation indicates that TRTRL is a valid method for performing temporal processing tasks.

Since it was very difficult to achieve low error rates, the simulation considerations de-

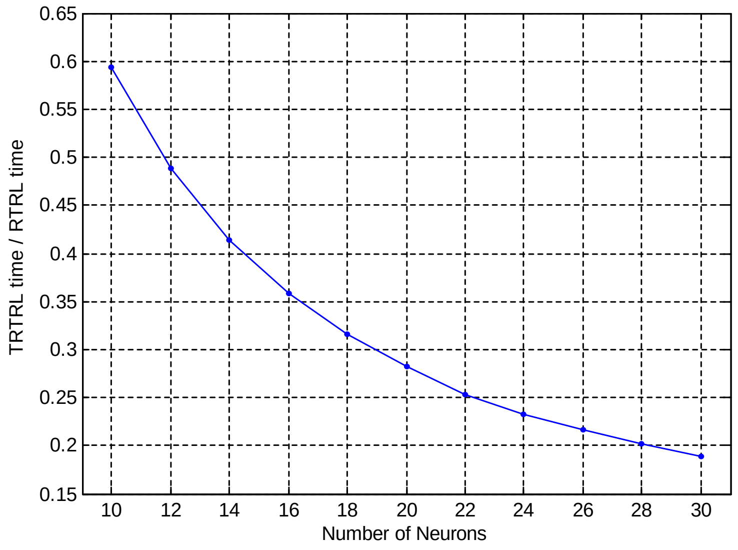

Figure 3-6: The ratio between TRTRL and RTRL training times for the Mackey-Glass (MG) chaotic time series prediction task.

scribed in section 3.2.1 were tested in numerous forms. The adaptive learning rate algorithm using individual learning rate parameters α_{ij} for each weight w_{ij} as well as a single parameter α were tested. The momentum term was also applied to the weight update rule alone, and also combined with the two variations on the adaptive learning algorithms. The highest performance, in terms of speed and error, was attained when the adaptive learning rate algorithm applied to a single α parameter was combined with a momentum term.

A significant benefit to TRTRL is the improvement in computation time. Because the number of calculations of the sensitivities is reduced, the computational speedup gain is by two orders of magnitude from $O(N^4)$ in RTRL to $O(N^2)$ with TRTRL. This speedup gain is reflected by the processing time ratio shown in figure 3-6. The time to train the networks when referencing the chaotic time series prediction task was recorded as the number of neurons was increased. As would be expected, the speedup gain for TRTRL increased substantially for larger networks.

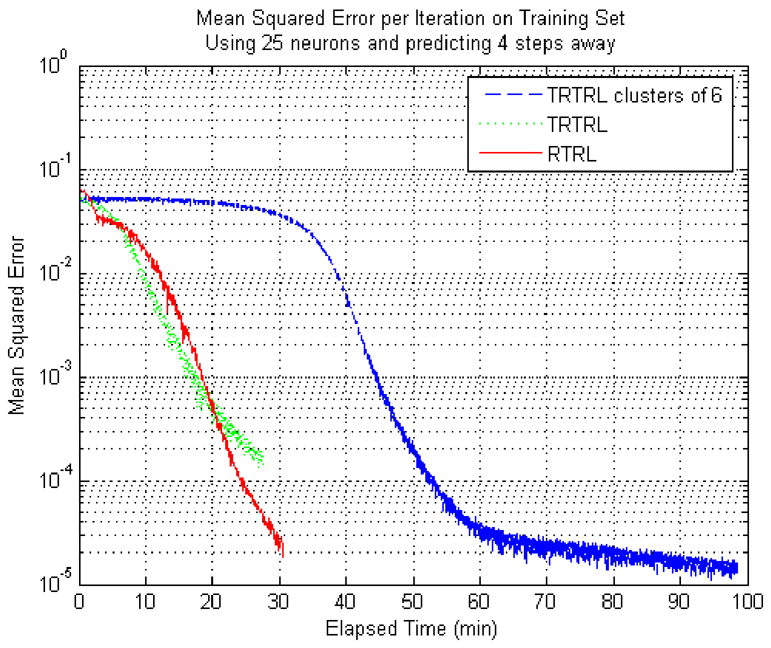

Figure 3-7: A comparison of RTRL, TRTRL, and clustered TRTRL performing the sequence memorization task.

3.2.4 Sequence Memorization

The final simulation performed was the sequence memorization task. The latter tests the ability of the neural network to store data for varying lengths of time. As the time to store data increases, also known as the memory depth, the network requires longer time to converge. This testbench showed the greatest amount of disparity between RTRL, TRTRL, and clustered TRTRL. Several memory depths were tested with all three algorithms. A memory of 4 was chosen because it was best able to show the difference between the algorithms. The test was performed with 25 neurons for RTRL, TRTRL and clustered TRTRL. The input was a long string of randomly generated sequences similar to the ones shown in Table 2.1.

As can be seen from Figure 3-7, TRTRL and RTRL reached an acceptable MSE of 10^{-3} at about the same time. This observation reinforces our earlier result that performance degradation between RTRL and TRTRL is minimal. What is interesting is that it takes about double the amount of time for TRTRL with clusters of 5 neurons to reach the same prediction accuracy with a MSE of 10^{-3}. The degradation in convergence rate of approximately 2, observed with clustered TRTRL, is generally acceptable. It is important to remember that clustered TRTRL requires far less memory and computational resources per neuron to update the gradient than does TRTRL.

Chapter 4

Scalable Hardware Architecture

4.1 Overview

We next introduce a novel hardware architecture for the scalable realization of TRTRL. There have been few attempts in the past to model a recurrent neural network in hardware, and none directly pertaining to RTRL. The biggest restriction is the immense storage requirements, which would have made it impractical to transcribe the algorithm to a FPGA. A programmable hardware device does not have as much memory available as a general purpose CPU, yet it is able to parallelize and perform calculations much quicker because it is designed with a specific algorithm in mind. With TRTRL, the storage requirements are reduced by two orders of magnitudes and with clustered TRTRL, the algorithm becomes far more localized. Given that each neuron (or processing entity, i.e. PE) only calculates the sensitivities associated with its ingress and egress links and only with those that are within its cluster, the memory requirements are much lower and the number of interconnects is greatly reduced. By exploiting these attributes in the design of a hardware architecture, a modern FPGA device (e.g. Altera Stratix II) will be able to hold at least 200 neurons, whereas a general purpose CPU will only be able to perform the algorithm at a practical rate with 30 neurons.

4.2 Network Architecture

The architecture of the network is based upon the clustered TRTRL scheme discussed above. To help illustrate the architecture design, we refer to Figure 4-1. This diagram is a more detailed extension of the clustered TRTRL algorithm depicted in Figure 3-2. The network described here is designed for implementation on a VLSI device, such as a FPGA. We next provide detailed functional description for each of the key component in the proposed design.

4.2.1 The Input Layer

As the input layer receives test patterns from the outside environment it sends them to each network cluster as well as to each output layer node, via a dedicated bus. Consequently, each cluster will forward the input values to each of its neurons/nodes.

4.2.2 The Cluster

The network is divided into 12 clusters containing 8 neurons each. A cluster also encompasses a controller and a dedicated memory, as well as provides the bias input and the sigmoid squashing functionality to its nodes (i.e. intra-cluster neurons). Each cluster hosts a dedicated bus line leading to each input layer neuron as well as to the main network controller. The main network controller provides a communication interface and buffer between each cluster and the output nodes in order to manage data flow. The cluster also hosts an intra-cluster bus which aids in cross-neuron communications. In order to properly handle the data between the hidden layer and the input and output layers, the cluster manages its own (small) memory space.

4.2.3 The Hidden Layer Neurons

Each hidden layer node communicates solely with the cluster controller in its communications with other neurons in the same cluster. Several pieces of data need to be shared between each cluster neuron: these are the ingress and egress sensitivities and the weights between each neuron i and output node o. The cluster nodes send and receive data on

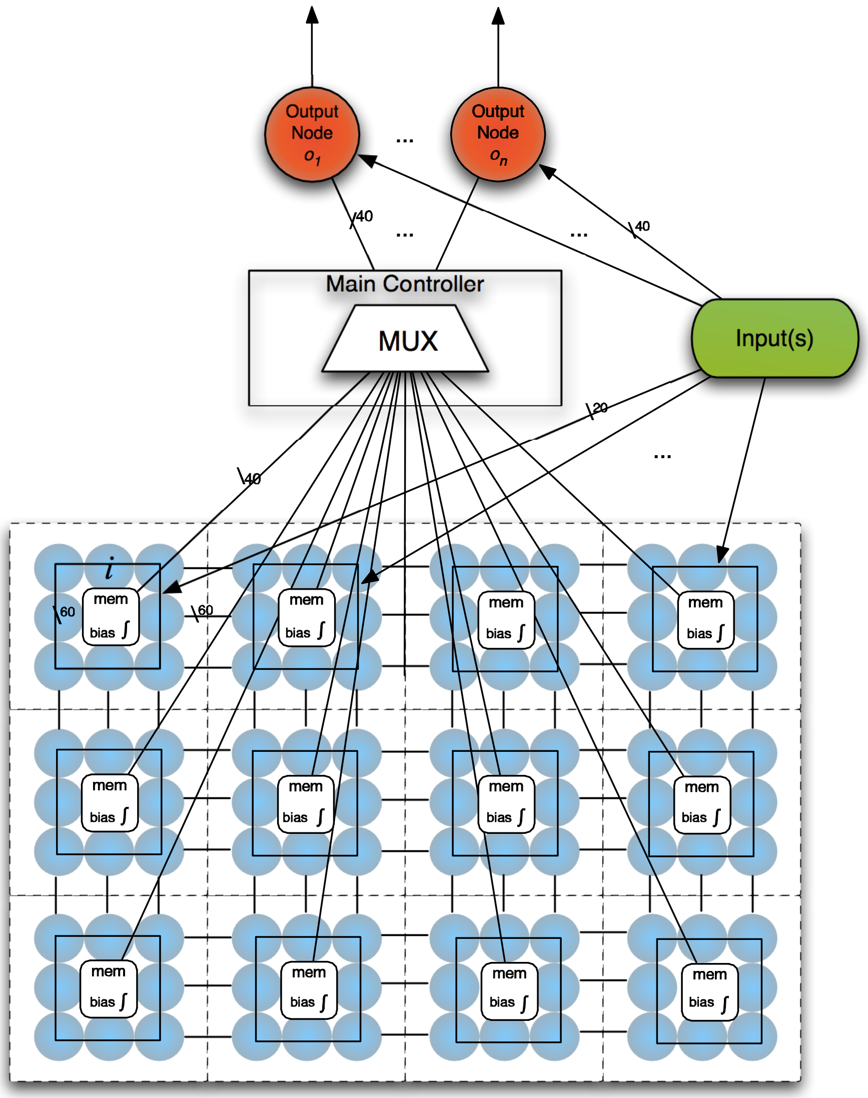

Figure 4-1: The hardware layout design that is used in transcribing the TRTRL algorithm to hardware.

the intra-cluster bus via a simple time division multiplexing (TDM) scheme, where every node either sends or receives data from the bus upon receiving its proper time-slot code. Given of the scalability attained with clustering, each neuron computes sensitivities, updates weights, and sends/receives data simultaneously over several time steps. Consequently, all calculations are performed in parallel across a single cluster B_i and also across the entire network.

Additionally, some neurons are connected to neurons that belong to another cluster. It is necessary to provide some inter-cluster connections in order to allow for information distribution and sharing within the internal states of the network. Nodes on the edges of each cluster B_i are connected to other nodes on the edges of another cluster, B_j.Note from Figure 4-1 that a node may have a maximum of 2 inter-cluster connections and these occur in nodes which lie on an inside corner of a particular cluster.

4.2.4 The Output Neurons

The output neurons are abstracted from the rest of the network through a main network controller which has a dedicated data buffer. Similarly to the TDM scheme employed by the cluster controller, the network controller allows clusters to read/write to the output neurons during a particular time slot. After the network controller receives the data from each cluster, it sends it to the desired output layer node. The output layer neurons only calculate their activations and errors, as the weights between hidden and output nodes, w_{oi} are stored at hidden node i. After the output nodes receive the sensitivities between itself and each hidden node, it performs the weight update and the entire process repeats.

4.3 Data Flow

We now present a more detailed procedure for training the neural network which is shown in Table 4.1. What follows below is a brief explanation of each step along with a more detailed complexity analysis.

Table 4.1: Communication protocol between the different neurons in a TRTRL-based network

Step	node i	node o
1	Inputs are broadcast to each node	
2	$z_i(t), w_{oi} \longrightarrow (j, o)$	
3	$p^i_{ij}(t) \longrightarrow (j)$	
4.1	$y_i(t+1) = f(s_i(t))$	$y_o(t+1) = f(s_o(t))$
4.2	$y_i'(t+1) = f^{'}(s_i(t))$	$y_o'(t+1) = f^{'}(s_o(t))$
5		$e_o(t) = y_o(t) - d_o(t)$
6		$e_o(t), f^{'}(s_o(t)) \longrightarrow (j)$
7	$p^o_{oi}(t+1)$	
8	$p^o_{oi}(t+1) \longrightarrow (o)$	
9	$p^i_{ji}(t+1)$	
10	$p^i_{ij}(t+1)$	
11	$p^o_{ij}(t+1)$	
12	$\Delta w_{ij} = \alpha e_o(t) p^o_{ij}(t+1)$	$\Delta w_{oj} = \alpha e_o(t) p^o_{oj}(t+1)$
	where $j, i \in [1, 2, ..., N]$ and $o \in$ output neuron set	

4.3.1 Protocol Explanations

The following is a detailed description of the protocol-flow performed by the TRTRL-based hardware network implementation. The steps correlate to those presented in Table 4.1.

1. Using the bus lines that connect the inputs to each node, the input values are broadcast to all nodes including the output node.

   Each node receives inputs from the input layer via a dedicated broadcast line to each cluster (see section 4.2.3). Assuming 20 bits per neuron and only one input per time step, each input node must send this value to B clusters. The individual cluster controller forwards the value to $\frac{N}{B}$ nodes, yielding a total of $\frac{N}{B}$ cycles + 1. Using the architecture in Figure 4-1, with 12 clusters of 8 neurons for a total of 96 hidden units, the broadcast consumes 9 clock cycles.

2. Node i broadcasts $z_i(t)$ and w_{oi} to all connected nodes j and all output neurons o.

   Node i sends $z_i(t)$ and w_{oi} to all of its intra-cluster nodes via the shared cluster bus. Further, node i sends the same data to its inter-cluster neighbors via dedicated lines, which are defined using a bitmask for each node.

3. Node i broadcasts $p^i_{ij}(t)$ to all connected nodes j.

   From steps 2 and 3 we note that each node must send 3 values totaling 60 bits to its intra-cluster neighbors. Therefore, we make the intra-cluster bus 60 bits wide.

   Node i sends $z_i(t)$, w_{oi} and $p^i_{ij}(t)$ to the cluster controller in a single transmission along a 60 bit bus. The cluster controller thereafter sends $z_i(t)$ and w_{oi} to output node o. This operation will require each node to send information to 7 neighboring nodes for a total of 56 clock cycles per cluster. Depending on whether a node is connected to an inter-cluster neighbor, a node will send the necessary information on its own.

4. Node i computes its activation and activation-derivative, $y_i(t+1)$ and $y'_i(t+1)$ respectively. Simultaneously, using the data received in step 2, each output neuron o computes the activations of the output neuron: $y_o(t+1)$ and $y'_o(t+1)$.

   All nodes, i and o, calculate their respective activations: $y_i(t) = f(s_i(t))$ or $y_o(t) = f(s_o(t))$, where

   $$s_i(t) = \sum w_{ij} z_j(t), \ j \in \text{ connected nodes.} \tag{4.1}$$

   This requires $\frac{N}{B}$ multiplications + $\frac{N}{B}$ summations yielding 16 cycles/node.

   Next, all nodes must access $f(\cdot)$, the sigmoid squashing function, which resides at the cluster level. Each cluster will have a small LUT with a piecewise linear implementation of the sigmoid described in detail in section 2.4.1. This requires1 memory access/node and is constrained by the speed of the sigmoid.

   All nodes are also required to calculate the derivative of the sigmoid, $f'(\cdot)$. Similarly, this operation requires 1 memory access/node and is also constrained by the speed of the sigmoid derivative.

5. Node o calculates the error $e_o(t)$.

   The output node o, calculates $e_o(t) = y_o(t) - d_o(t)$. This is a simple subtraction operation yielding 1 cycle.

6. Node o broadcasts $e_o(t)$ and $f^{'}(s_o(t))$ to all nodes j.

   The output node broadcasts $e_o(t)$ and $f'(s_o(t))$ to each cluster along a dedicated 40 bit wide bus. The cluster controller then transmits each value to each respective intra-cluster neuron over its 60 bit bus using the 40 least significant bits. Theoretically, this takes a single clock cycle to broadcast to the cluster, and $\frac{N}{B}$ cycles to get to each node. Additionally, some extra cycles will be spent in the multiplexing operation of the main controller.

7. The ingress sensitivity of the output nodes are calculated at node i, and output nodes do not have egress sensitivities.

   These sensitivities are derived from eq. (3.1) by utilizing the relationship $p^o_{oi}(t+1) = f'_o(s_o(t))[w_{oi}p^i_{oi}(t) + z_i(t)]$. The latter can only be calculated once node i receives $f'_o(s_o(t))$ from node o, which happens during Step 6. This operation consumes 2 multiplications + 1 addition.

8. Node i sends $p^o_{oi}(t)$ to output neuron o.

   The ingress sensitivities of the output node to node i ($p^o_{oi}(t+1)$) are sent to the output nodes from each node i. This requires $\frac{N}{B}/2$ transmissions/cluster resulting in $\frac{N}{2}$ total transmissions to the output node assuming B transmissions happen all at once. We can do $\frac{N}{2}$ because the bus is 40 bits wide and each value is only 20 bits long.

9. Calculate egress sensitivities at node i: $p^i_{ji}(t+1) = f'_i\left(s_i\left(t\right)\right)\left[w_{ij}p^j_{ji}(t) + \delta_{ji}y_i(t)\right]$.

   The cluster controller continues performing Step 8 while nodes within the cluster calculate the egress sensitivities $p^i_{ji}(t+1)$. Each neuron must perform a total of: (2 multiplications + 1 addition and a comparator)$\cdot\frac{N}{B}$ nodes/cluster.

10. Calculate ingress sensitivities at node i: $p^i_{ij}(t+1) = f'_i\left(s_i\left(t\right)\right)\left[w_{ij}p^j_{ij}(t) + z_j(t)\right]$.

    All nodes calculate the ingress sensitivities $p^i_{ij}(t+1)$. Each neuron must perform a total of: (2 multiplications + 1 addition)$\cdot\frac{N}{B}$ nodes/cluster.

11. Calculate output sensitivities at node i: $p^o_{ij}(t+1) = f'_o\left(s_o\left(t\right)\right)\left[w_{oi}p^i_{ij}(t) + w_{oj}p^j_{ij}(t) + \delta_{io}z_j(t)\right]$.

    Each node calculates its sensitivity to the output node o, $p^o_{ij}(t+1)$. This calculation takes (3 multiplications + 3 additions + 1 comparator)$\cdot\frac{N}{B}$ nodes/cluster at each neuron, and must also be performed at the output layer nodes.

* Since the entire network is localized, Steps 9-11 are performed concurrently at each node i.

12. Perform weight-update at each neuron i and output neuron o.

    Using the just-calculated sensitivities, each neuron i and o updates its weights by (a) calculating the gradient and (b) performing the actual update:

    (a) Calculate the gradient: $\Delta w_{ij} = \alpha e_o(t) p^o_{ij}(t)$. As described earlier in section 2.4.2 we will use a value of a power of 2 for α. The aggregate number of cycles results in 1 shift + 1 multiplication/neighbor, yielding ($\frac{N}{B}$ + the number of inter-cluster neighbors) total calculations.

    (b) Perform the weight update: $w_{ij}(t+1) = w_{ij}(t) + \Delta w_{ij}$. This requires $\frac{N}{B}$ + the number of inter-cluster neighbors addition operations.

4.3.2 Other Considerations

We also need to consider the memory consumption per neuron, as discussed in section 3.1.2. Several 20-bit wide dual-port block RAMs will be allocated to each node. Using the network architecture diagram presented in Figure 4-1, in which we have 8 neurons per cluster with a maximum of 10 neighbors, each dpRAM block will contain 10 addresses.

4.3.3 Implementation of Sigmoid and Sigmoid Derivative

The PWL approximation of the sigmoid and the sigmoid derivative functions described in Section 2.4.1 is employed. The companding architecture in [33] can be adapted to the PWL sigmoid implementation. First, the input signal is compared to all of the 15 possible intervals of the sigmoid in parallel. The actual interval is determined by using the XOR function. Once the interval is determined, the output is computed using shifts and adds for the interval that the input value lies in. This scheme employs only combinatorial logic and does not require a clock, as it performs the entire process in a single clock cycle.

Table 4.2: Preliminary hardware design results.

	Used	Available
Logic Blocks (Adaptive Lookup Tables)	862	48,352
Memory	1000 bits	2.5 Mbit
DSP Blocks	51	288
Clock Speed	32.58 MHz	(30.694 ns)

4.4 Preliminary Results

A simplified TRTRL algorithm was designed using VHDL and was targeted for the Altera Stratix II EP2S60 FPGA device. The preliminary design included one neuron block, one sigmoid block and one sigmoid derivative block. The majority of the computations take place in these three IP blocks and therefore they provide a good estimate of the total resource requirements that will be needed for a full network implementation. The cluster block contains only glue and communication logic and the output neuron block has a function that is only slightly different from hidden layer neurons.

The design was synthesized using the Altera Quartus design tools and the chip resource requirements are shown in Table 4.2. Referring to the Table, the amount of ALUTs used was less than 2% of the total logic resources available on the FPGA and the amount of memory used was less than 1% of the total available memory. The limiting factor in the results lies in the DSP block 9-bit elements required, which amounted to 17% of the total available on chip. This is a result of using an external, generalized fixed point multiplier library which is not targeted for this chip. Using the design as is, at best it is possible to fit 5 neurons on the chip.

There are several options to reducing the usage of these DSP blocks. One method is to use on-chip multipliers or to use a less resource intensive approach such as a Booth multiplier which can fit in the available ALUTs. Based on the current design, the 5 neurons will only take up about 10% of the logic and memory blocks on the chip leaving a large amount of room for multipliers.

The design can also be optimized based on the clock speed. Because the given clock speed is high, the neurons can be multiplexed where one neuron will act like several neurons. The individual components can compute their activations and sensitivities and store them.

Next, the same neuron blocks can calculate another set of activations and sensitivities using different external inputs. In the end, the network will operate slower, but with less resource-intensive blocks in use.

Chapter 5

Conclusions

5.1 Summary of Thesis Contributions

This thesis presented a novel approach for reducing the resource requirements of the RTRL network algorithm to $O(N^2)$ while retaining high-performance. The method is based on limiting the sensitivities for each neuron to weights associated with either incoming or outgoing links. Based on standard testbench scenarios, it is demonstrated that the performance degradation is kept low while speed and storage requirements are significantly reduced.

Moreover, the resulting network architecture easily lends itself to hardware implementations as it is highly localized. Using a clustered network structure for improved scalability and efficiency, we verified that it is feasible to transcribe such an RNN to hardware. Using hardware implementation, it is now much more feasible to construct and train large recurrent neural networks comprising of hundreds of nodes. Moreover it is possible to efficiently connect several networks together to achieve even greater scalability.

5.2 Future Work

There are several directions of future work that can easily leverage the results described in this thesis. First, the learning concept assumed was straight-forward gradient descent, which has significant drawbacks in terms of convergence rate. This is true in the context of general optimization tasks and not specifically tied to neural networks. To that end, future work should be focused on investigating more advanced gradient-based learning schemes, such as those employing second-order derivative information to improve convergence properties.

Another important aspect that merits research attention has to do with resolving the bottlenecks associated with backpropagating the network error to the hidden layer nodes. In the hardware simulations, this played the role of the scalability limiting factor, since it violates the locality of the algorithm. It may also be possible to improve the communication protocol to streamline the network learning process such that a more localized process is obtained.

Applying the hardware network architecture to a real-time problem will be a very interesting undertaking. Because we are able to achieve a much higher neuron density in silicon, it is now feasible to apply the network to reinforcement learning tasks that have a very large state-space. The clustered TRTRL architecture is highly scalable, thus it may also be useful to build and connect together a large array of such networks in order to apply them to highly complex stochastic learning problems.

5.3 Relevant Publications

The core ideas pertaining to TRTRL, as presented in this thesis, have appeared in the following publication:

- D. Budik, I. Elhanany, "TRTRL: A Localized Resource-Efficient Learning Algorithm for Recurrent Neural Networks," 2006 IEEE International Midwest Symposium on Circuits & Systems (MWSCAS), San Juan, Puerto Rico, August, 2006

Bibliography

Bibliography

[1] M. Boden, “A guide to recurrent neural networks and backpropagation,” 2001, school of Information Science, Computer and Electrical Engineering. Halmstad University, Sweden.

[2] R. W. Williams and K. Herrup, “The control of neuron number,” *The Annual Review of Neuroscience*, no. 11, pp. 423–453, 1988.

[3] M. Jordan, “Serial order: A parallel distributed processing approach,” Institute for Cognitive Science Report 8694. University of California, San Diego, 1986.

[4] S. Haykin, *Neural Networks: A Comprehensive Foundation*, 2nd ed. Upper Saddle River, NJ: Prentice-Hall, 1999.

[5] J. L. Elman, “Finding structure in time,” *Cognitive Science*, no. 14, pp. 179–211, 1990.

[6] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, “Learning internal representation by error propagation,” in *Parallel Distributed Processing: Explorations in the Microstructure of Condition*, D. E. Rumelhart and J. L. McClelland, Eds. Cambridge, MA: MIT Press, Bradford Books, 1986, vol. 1, pp. 318–362.

[7] R. J. Williams and D. Zipser, “Gradient-based learning algorihtms for recurrent networks and their computational complexity,” in *Backpropagation: Theory, Architectures, and Applications*, Y. Chauvin and D. E. Rumelhart, Eds. Hillsdale, NJ: Lawrence Erlbaum, 1995, pp. 433–486.

[8] P. J. Werbos, “Beyond regression: new tools for prediction and analysis in the behavioral sciences,” 1974, unpublished doctoral dissertation. Harvard University.

[9] P. Werbos, "Backpropagation through time: what it does and how to do it," *Special issue on neural networks, Proceedings of IEEE*, vol. 78, pp. 1550–1560, October 1990.

[10] A. J. Robinson and F. Fallside, "The utility driven dynamic error propagation network," Cambridge University Engineering Department, Cambridge, England, Tech. Rep. CUED/F-INFENG/TR.1, 1987.

[11] R. J. Williams and J. Peng, "An efficient gradient-based algorithm for on-line training of recurrent network trajectories," *Neural Computation*, vol. 2, pp. 490–501, 1990.

[12] B. A. Pearlmutter, "Learning state space trajectories in recurrent neural networks," *Neural Computation*, vol. 1, pp. 263–269, 1989.

[13] R. J. Williams and D. Zipser, "A learning algorithm for continually running fully recurrent neural networks," *Neural Computation*, no. 1, pp. 270–280, 1989.

[14] J. Schmidhuber, "A fixed size storage o(n3) time complexity learning algorithm for fully recurrent continually running networks," *Neural Computation*, vol. 4, pp. 243–248, 1992.

[15] G.-Z. Sun, H.-H. Chen, and Y.-C. Lee, "Green's function method for fast on-line learning algorithm of recurrent neural networks." in *NIPS*, 1991, pp. 333–340.

[16] D. Zipser, "A subgrouping strategy that reduces complexity and speeds up learning in recurrent networks," *Neural Computation*, no. 1, pp. 552–558, 1989.

[17] N. Euliano and J. Principe, "Dynamic subgrouping in RTRL provides a faster o(n2) algorithm," in *Acoustics, Speech, and Signal Processing, 2000. ICASSP '00. Proceedings. 2000 IEEE International Conference on*, vol. 6, Istanbul, 2000, pp. 3418–3421.

[18] K. P. Unnikrishan and K. P. Venugopal, "Alopex: a correlation-based learning algorithm for feedforward and recurrent neural networks," *Neural Computation*, vol. 6, pp. 469–490, 1994.

[19] K. W. Ku, M. W. Mak, and W. C. Siu, "Recurrent neural network with backpropagation through time for speech recognition," in *ICNNSP (1995)*, China, Dec 1995.

[20] M. W. Mak, K. Ku, and Y. L. Lu, "On the improvement of real time recurrent learning algorithm for recurrent neural networks," *Neurocomputing*, vol. 24, pp. 13–36, 1999.

[21] M. Henon, "A two-dimensional mapping witha strange attactor," *Comunications in Mathematical Physics*, vol. 50, pp. 69–97, 1976.

[22] P. Murtagh and A. Tsoi, "Implementaion issues of sigmoid function and its derivative for vlsi digital neural networks," *IEE Proceedings*, vol. 139, no. 3, pp. 207–214, May 1992.

[23] S. Draghici, "On the capabilities of neural networks using limited precision weights," *Neural Networks*, vol. 15, pp. 395–414, 2002.

[24] R. Lyon, "Two's complement pipeline multipliers," *Communications, IEEE Transactions on [legacy, pre - 1988]*, vol. 24, no. 4, pp. 418–425, Apr. 1976.

[25] D. Bishop, "Fixed point package user's guide," IEEE Proposed VHDL library extensions. http://www.vhdl.org/vhdl-200x/vhdl-200x-ft/packages/files.html, 2006.

[26] J. Elsner, "Predicting time series using a neural network as a method of distinguishing chaos from noise," *J. phys. A : math. gen*, vol. 25, no. 4, pp. 843–850, 1992.

[27] U. Konur and A. Okatan, "Time series prediction using recurrent neural network architectures and time delay neural networks," *Transactions on Engineering, Computing and Technology*, vol. 3, December 2004.

[28] M. Mackey and L. Glass, "Oscillation and chaos in physiological control systems," *Science*, vol. 197, pp. 287–289, July 1977.

[29] R. S. Crowder, III, "Predicting the mackey-glass timeseries with cascade-correlation learning," in *D. Touretzky, G. Hinton and T. Sejnowski eds., Connectionist Models Summer School*, Carnegie Mellon University, 1990, pp. 117–123.

[30] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery, *Numerical Recipes in C: The Art of Scientific Computing*, 2nd ed. New York: Cambridge University Press, 1992.

[31] N. Kamiyama, N. Iijima, H. Mitsui, Y. Yoshida, and M. Sone, "Tuning of learning rate, momentum and network scale on backpropagation," in *Intelligent Control and Instrumentation, 1992. SICICI '92. Proceedings., Singapore International Conference on*, vol. 1, Feb. 1992, pp. 384–388.

[32] V. Plagianakos, D. Sotiropoulos, and M. Vrahatis, "An improved backpropagation method with adaptive learning rate," in *2nd International Conference on Circuits, Systems, and Computers*, Piraeus, Greece, 1998.

[33] O. Arazi and I. Elhanany, "A scalable architecture for high-speed digital companding," in *IEEE 48th Midwest Symposium on Circuits and Systems*, August 2005.

Vita

Daniel Budik was born in Korosten, Ukraine, near Kiev on May 17, 1983. He and his family moved to Knoxville, TN in 1992. After graduating from Webb High School in 2001, he attended The University of Tennessee where he received his Bachelor of Science degree in Computer Engineering in May 2005. He has continued his studies at the University of Tennessee where he received his Master of Science in Electrical Engineering in December 2006.