FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

FPGA implementation of a LSTM Neural Network

José Pedro Castro Fonseca

Integrated Masters in Electrical and Computer Engineering

Supervisor: João Canas Ferreira

Second Supervisor: Ivo JPM Timóteo

July 25, 2016

Resumo

Esta dissertação tem como objetivo a implementação em FPGA de uma Rede Neuronal *Long Short-Term Memory* (LSTM).

As redes neuronais são uma das técnicas mais usadas na área da Aprendizagem Computacional Profunda. Esta rede em particular, a LSTM, é uma rede recursiva, na medida em que a saída da cada neurónio, num instante de tempo, *também* serve de entrada no instante de tempo seguinte, e os seus elementos de memória conseguem recordar padrões em sequências de dados, apresentando vantagens claras, neste campo, face a redes neuronais convencionais.

As aplicações das Redes Neuronais são inúmeras, como se poderá atestar no capítulo que faz o levantamento do estado-da-arte. Embora a implementação em *software* seja a solução comum para todas estas arquitecturas, as implementações em *hardware* são ainda poucas e os benefícios do paralelismo inerente a uma plataforma de *hardware* dedicado não são aproveitados. É nesse enquadramento que este trabalho se posiciona, apresentando uma implementação inédita de LSTM em FPGA, fazendo apenas uso de recursos internos à mesma, apresentando um *speed-up* de 251 vezes face a uma implementação *software* — executada num Computador de alta *performance* — assim como a proposta de um circuito auxiliar que permita fazer o treino *on-chip* da rede, usando um método de perturbações estocásticas simultâneas (SPSA).

Abstract

The objective of this Dissertation work is to implement a Long Short-Term Memory (LSTM) Neural Network in an FPGA platform.

Neural Networks are one of the most commonly used techniques in Deep Learning. This particular type of network, as LSTM Network, is a recursive network, given that the neuron outputs in a given time are *also* fed as the input in the next time step and, that way, their memory elements can make sense of patterns within data sequences, unlike conventional neural networks.

This type of network, as well as many other kinds of networks, have been profusely implemented in software, and their practical applications are plentiful, as one can attest in the State of the Art chapter. However, there are only few FPGA implementation of these algorithms, and the benefits of the inherent parallelism offered by a dedicated hardware platform are not availed. This work, then, implements an LSTM Network in FPGA, using only its internal resources, and achieving a $251\times$ *speed-up* in processing time when compared to a software implementation, running on a high-end Desktop. A circuit that will allow on-chip training for this network, using simultaneous stochastic perturbations (SPSA), is also proposed.

Agradecimentos

Agradeço a todos os professores que se cruzaram comigo na minha vida e que, através da sua ação, me transmitiram os conhecimentos necessários para me formar como Engenheiro e produzir este trabalho, que espero que seja de uso para a comunidade científica e para a Humanidade. Uma palavra especial ao meu orientador deste trabalho, o Prof. João Canas Ferreira, que desde o primeiro momento acolheu esta proposta de trabalho e soube ver nela o seu potencial devido, e ao co-orientador, o Ivo Timóteo, que mais do que minha referência para a parte teórica deste trabalho, é um bom amigo.

Agradeço todo o esforço da comunidade *open-source* que desenvolveu grande parte das ferramentas com que trabalhei, desde o LaTeX em que escrevi este documento, ao Python e Numpy usado na automatização de tarefas e no teste duma primeira versão software da rede. Agradeço também ao autor das Circuit Macros, J.D. Aplevich, que muito foram úteis para a elaboração de quase todos os diagramas deste trabalho.

Uma última palavra de apreço para as pessoas especiais da minha vida. Os meus pais, que me formaram como Pessoa e me incutiram força em todos os momentos da vida, do qual esta tese não é excepção, e à minha namorada, a Mafalda, que apesar de não ser da minha área, sempre se entusiasmou e interessou por todos os progressos que fui atingindo ao longo destes meses.

José Pedro Castro Fonseca

"When you have exhausted all possibilities, remember this - you haven't."

Thomas Edison

Contents

List of Figures

Abbreviations and Symbols

ANN	Artificial Neural Networks
BPTT	Backpropagation Through Time
CA	Cellular Automata
CNN	Convolutional Neural Network
CPU	Central Processing Unit
DMA	Direct Memory Access
FPGA	Field-Programmable Gate Array
LFSR	Linear-Feedback Shift Register
LSTM	Long Short-Term Memory
PRNG	Pseudo-random Number Generator
RNN	Recursive Neural Networks
SPSA	Simultaneous Perturbation Stochastic Approximation

Chapter 1

Introduction

In this work, a hardware implementation of an LSTM Neural Network is presented, along with a proposal for a digital circuit that performs on-chip training. In the reference example, the proposed hardware network was 251 times faster than the reference software implementation running in a high-end Desktop. The training convergence, although simulated, was not conclusive.

1.1 Background

Artificial Neural Networks are one of the most popular algorithms in the field of Machine Learning. As the name suggests, their operation is inspired by the operation of the building blocks of our brain, the **neurons**. In spite of its high performance, one of their shortcomings is the fact that they cannot retain temporal dependences in a stream of data samples, thus not being suitable to process time-series data, such as audio, video or other kinds of time-varying data streams, where inputs at a given time step have a high temporal dependence with previous and future inputs.

To address this issue, several algorithms have been used, such as Hidden Markov Models (HMM) or Recurrent Neural Networks (RNN), but both of these methods fail to recall temporal dependences that extend over a large period of time, for the reasons that we will understand in Sections 2.1.3 and 2.1.4. In 1997 Hochreiter et al. proposed a novel RNN structure [4], the Long Short-Term Network (LSTM), where a memory cell was introduced, and the input/output/read-/write access is controlled by individual gates that are activated both by the incoming data samples, but also by the outputs from the previous time-step (it is an RNN after all). They are one of the state of the art methods in Deep Learning nowadays, as we can attest in Section 3.1.

1.2 Motivation

Hitherto, and to the best of my knowledge, all of the published applications that use LSTM are software based. But the parallel nature of the structure hollers for a dedicated hardware realization that can dramatically increase its performance, something that has only recently been done once [5] (see Section 3.2 for further details), and although it improves the processing time when

compared to a naïve software solution, it makes use of external memory to store the network parameters, which undermines the network performance. Furthermore, the design is not flexible to accommodate networks of different sizes, and lacks the ability to perform on-chip learning.

All these techniques are generally implemented in mainstream processors, making use of general high-level or low-level languages, where all the real parallelism is limited to the number of simultaneous threads that we can run on each physical core, which up to now generally have between 2-8 cores (mobile devices and general use personal computers),

In order to parallelize the computations to the fullest extent, a solution is to port it both to a Graphics Processing Unit (GPU) or even a Field-Programmable Gate Array (FPGA), but the porting process is not entirely automatic and to have the least performance drop possible, it has to be explicitly programmed in CUDA/OpenCL for GPUs, and a Hardware Description Language for FPGAs, with an increasing level of complexity and low-level details. Therefore, it is necessary to provide frameworks that can allow an FPGA to quickly reconfigure itself to run these kind of networks on demand, for a particular task that requires them, achieving a lower computation time, and unburdening the CPU from running it, thus saving performance for running other tasks related with the Operating System, for instance. Furthermore, when processing an incoming stream of highly dimensional data, or with high throughput, a CPU solution might not be scalable, and could benefit greatly from a dedicated hardware implementation.

1.3 Objectives

Taking into account the considerations done in 1.2, a hardware implementation of an LSTM Network is proposed, *with on-chip learning*, improving the performance, capability and flexibility of the existing solution of [5]. Moreover, the HDL description of the network is parameterizable, meaning that to synthesize a network for a given size, we only need to change the appropriate parameter at synthesis time, therefore requiring no redesign effort whatsoever, unlike [5].

1.4 People Involved

Besides myself, the candidate to the Master's Degree, there are two more people involved, namely

- **Supervisor** – The Professor João Canas Ferreira, auxiliary professor at the Faculty of Engineering of the University of Porto.
- **Second Supervisor** – Ivo Timóteo, MSc, a Computer Science PhD candidate at Cambridge University, UK, in the field of Artificial Intelligence.

1.5 Overview of the Document

In Chapter 2, the theoretical foundations will be laid out. Section 2.1.1 presents the basic concepts of Machine Learning, and in the following sections, the theoretical details of ANNs, RNNs and

LSTMs. Section 2.1.4.2 provides a quick explanation of the training algorithm that will be used in the final solution, which will be outlined in Section 4.3.2.

In Chapter 3, the state of the art of LSTMs, their applications (Section 3.1), their hardware implementations (Section 3.2) and the current work regarding the chosen training algorithm (Section 3.3) are presented.

In Chapter 4 the proposed architecture of the hardware LSTM network is explained in detail, as well as details related to each of its constituent blocks.

Finally, in Chapter 5, I present the results achieved with the hardware network of Chapter 4.

Chapter 2

Problem Characterization

2.1 Theoretical Background

2.1.1 Basic Concepts of Machine Learning

Machine Learning is a field of Computer Science that studies the development of mathematical techniques that allow software to learn autonomously, without an explicit description of each rule of operation. Its goal is to extract latent features from the data that allow an immediate classification of each input data into a particular class – the catch is that there is no previous rule formulation, but instead we have an adaptive model that adjusts is parameters according to the input data it receives, improving the estimates it yields as it receives new input samples.

Let us consider a practical example. For instance, suppose we want to build a program that given an input audio waveform representation of a spoken word, it matches it into a particular word of a dictionary. We could, of course, devise a set of rules and exceptions for each word analysing some of its features (perhaps the Fourier representation of each one, and, from it, manually finding the appropriate rules for each), but apart from being a very complex task, it wouldn't be a scalable solution, given the enormous number of words in each language. The approach taken by Machine Learning is different, and instead of manually processing each waveform, we build a large dataset, of size N, containing the waveforms of several words $[\mathbf{x}_1, \mathbf{x}_2, \mathbf{x}_2, \ldots, \mathbf{x}_N]$ – we call this dataset the **training dataset** – and we feed it to our model. Each of the i-th data point was previously labelled, and in fact we feed each training data point $\mathbf{x}_i$ *along* with its corresponding label t_i, so that the model can adapt its parameters accordingly to the *target value* it is supposed to classify. This set of labels $\mathbf{t} = [t_1, t_2, t_3, \ldots, t_N]$ is called the **target vector**.

We are, then, left with the following question: how can the model quantitatively evaluate the quality of its current set of parameters? That could be achieved in a number of ways, but the most usual is using a **Cost Function** that, as the name suggests, measures the cost of each wrong classification of the model. The model then evolves in a way that minimizes the cost function. A usual choice for the cost function is the **sum of squares error**, given the Gaussian

Noise assumption. Mathematically, if $y_\theta^i = y_\theta(x_i)$ is the prediction for the input data point x_i with label t_i, given the current set of parameters θ, the cost function using this metric is given by

$$J(\theta) = \frac{1}{2}\sum_{i=1}^{N}\left(y_\theta^i - t_i\right)^2. \tag{2.1}$$

Sometimes, instead of applying the raw data to the model, we can apply some sort of *preprocessing* to the data to extract the relevant features from it. For instance, instead of just feeding a raw image, we can perform several operations like edge detection or low-pass filtering, and apply them in parallel. In cases of highly dimensional data (i.e. each data vector has a very high number of features), we can apply techniques like **Principal Component Analysis** to reduce the feature space to a smaller dimension one, where the previous features were combined into two or three new features that pose themselves as the most relevant.

The problems described above are, in fact, a subset of the problems that Machine Learning tries to address. These problems are called **classification problems**, because for each input data point, our model tries to fit it into the most appropriate class. But we can also address **regression problems** where the output is not limited to a discrete set of values but rather a continuous interval. On the other hand, the Neural Network that this work will implement addresses a special kind of classification problem, where the classification decision is influenced not only by the current input sample, but also by a given *window of samples* that trail the current sample.

In summary, the most typical setting for a Machine Learning problem is having a large *input dataset* which we use to *train* our model (i.e. allowing him to dynamically adapt its set of parameters θ), in order to produce an output label y_i for each of them that minimizes a quality metric, the *Cost Funcion*, which can be chosen to the sum of squared differences, the log-loss, or any other appropriate mathematical relationship between the estimate y_i and the correct label t_i.

Now that the basic Machine Learning concepts have been presented, I will discuss, henceforth, one of the most important algorithms that address the supervised classification problems, the *Artificial Neural Networks* that will be discussed in Section 2.1.2 as a contextual introduction to the main theme of the thesis, which will be Recurrent Neural Networks (Section 2.1.3), namely the **Long Short-Term Memory Networks** (Section 2.1.4), both of which are improvements over the initial formulation of the ANNs. These two last networks branch even further from these set of problems, and are usually employed in *Deep Learning* tasks, where we try to extract even higher level information from data at the expense of increased model complexity.

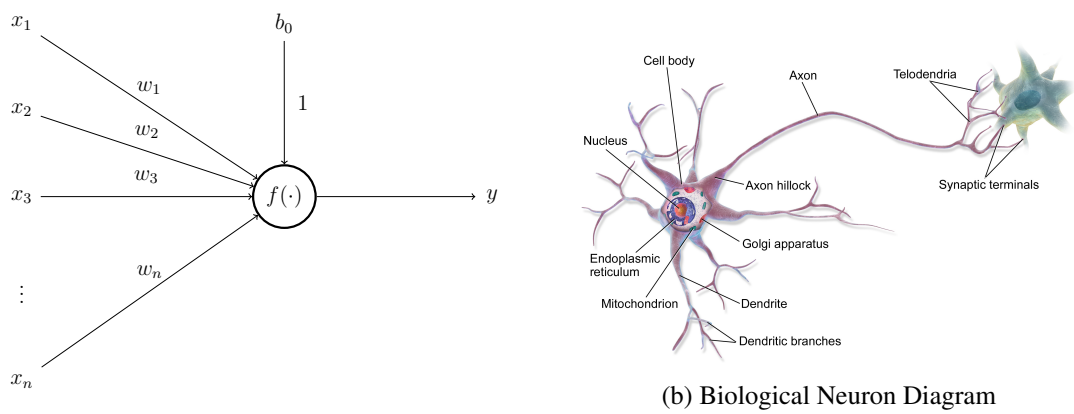
2.1.2 Artificial Neural Networks

Artificial Neural Networks (ANN) are mathematical structures that, as the name suggests, try to mimic the basic way of how a human brain works. ANN's building blocks, like their biological counterpart, model the high-level behaviour of biological neurons, in the sense that they neglect unnecessary biological aspects (such as modeling all the voltages across the neuron and all its electromagnetic interactions), and only retain its fundamental underlying mathematical function,

which is a weighted linear combination of its inputs subject to a *activation function*, i.e. a function that outputs a decision value depending on its inputs. Mathematically, we have

$$y = f(\mathbf{w}^T\mathbf{x}) \tag{2.2}$$

where $\mathbf{w} = [w_0\ w_1\ w_2\ w_3\ \dots\ w_n]$ is the input weight vector, $\mathbf{x} = [x_0\ x_1\ x_2\ x_3\ \dots\ x_n]$ the input vector, b_0 is the bias factor and $f(\cdot)$ is the chosen activation function. Furthermore, we call the scalar quantity $a = \mathbf{w}^T\mathbf{x}$ the **activation**, since its value determines how the activation function will behave. Figure 2.1 exemplifies the roles of these variables within our neuron model, and compares each part of it with the biological counterpart.



(a) Neural Network Node

(b) Biological Neuron Diagram

Figure 2.1: In Figure 2.1a. each input feature x_i is weighted by its corresponding weight w_i. During the training procedure, these weights are adjusted so that the output y approaches the target value. In Figure 2.1b, we see the diagram of an actual multi polar neuron. The dendrites, where the stimuli are received, plays a role similar to that of the input nodes. The axon transmits the signal to the synaptic terminals, that are similar to the y output. Source: Wikipedia

As far as the activation function is concerned, we can have several types. An immediate choice would be the **Binary Step Function** that outputs -1 if the activation is **below** a given threshold and 1 otherwise. There can also be **real valued activation functions**, whose output is not binary, but rather that of a continuously differentiable function, such as the logistic sigmoid $\sigma(a) = \frac{1}{1+e^{-a}}$ or the hyperbolic tangent $\tanh(a)$. This aspect will prove useful for the usual training methods, that involve the computation of derivatives. In Figure 2.2 these activation functions are plotted.

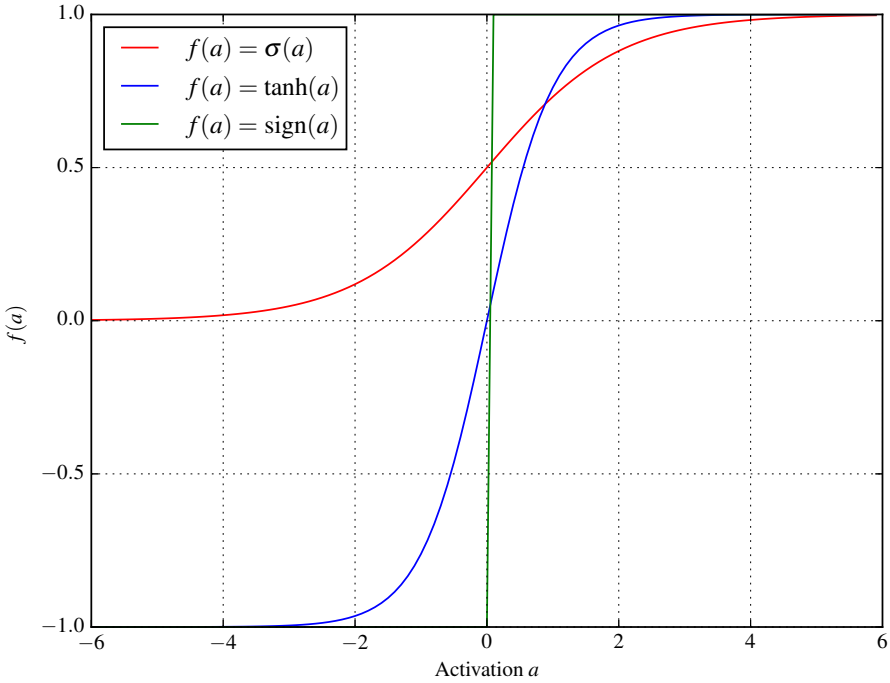


Figure 2.2: Three different activation functions. As you can see, the hyperbolic tangent has the same extreme values as the sign step function, but has a smooth transition between them, which can be interpreted as a *soft decision* in the more ambiguous middle region, reflecting the degree of uncertainty on the decision. On the other hand, the sigmoid function goes from zero to one, and is also smooth like the hyperbolic tangent.

A neuron by itself can be thought of as a simple linear regression, where we optimize the weight of each feature according to a target value, or function. While important in some applications, the main interest in ANN is to evaluate increasingly more complex models, and not a simple linear regression. This is achieved by *chaining* nodes to one another, connecting the output of a given node to one of the inputs of another. We call *layers* to a group of these nodes that occupy the same hierarchical position. There can be any number of layers, with any number of nodes, but most implementations generally have 3 layers: the *input* layer, the *hidden* layer (in the middle) and the *output* layer. Figure 2.3 suggests a possible structure for a 3 layer ANN.

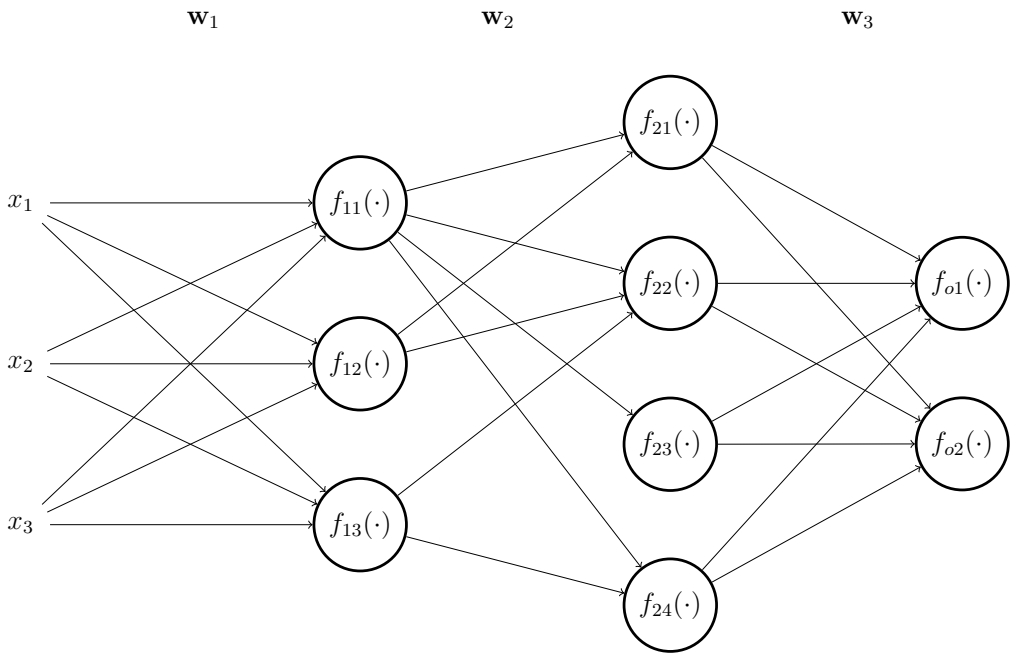


Figure 2.3: A three layer ANN. We have omitted some of the connections in the hidden layer, for simplification purposes. $\mathbf{w}_1$ represents the weight matrix of the input layer, $\mathbf{w}_2$ the weight matrix of the connections between the input layer and the hidden layer, and $\mathbf{w}_3$ the weight connections between the hidden and the output layer. $f_{ij}(\dots)$ is the activation function of the j-th neuron of the i-th layer. Since they can be different, I chose different indexes to each.

Regarding the training of ANNs, it is performed through a two-step process: first, a **feed-forward** step where the input is applied, and the activations are evaluated in succession up to the output neurons; then, we perform the **backpropagation** step, where we calculate the errors in each of the nodes (the so-called **deltas** of equation 2.5), but now from the output to the input: the weights are updated and optimized using an iterative method called *Gradient Descent*, where if τ is the current time step, the next update on the weight matrix $\mathbf{w}^{(\tau+1)}$ is given by

$$\mathbf{w}^{(\tau+1)} = \mathbf{w}^{(\tau)} - \eta \nabla E(\mathbf{w}^{(\tau)}) \tag{2.3}$$

where $E(\cdot)$ is the error function. As we can see, the weight matrix is moved in the direction that minimizes the error function the most, and η controls how fast this is achieved, being the reason why it is called the **learning rate**.

The computation of gradient of the error function comprises the evaluation of its derivatives with respect to each weight of all network connections, w_{ij}. They are

$$\frac{\partial E}{\partial w_{ji}} = \delta_j f(a_i) \tag{2.4}$$

where $f(\cdot)$ is the activation function of the neuron and

$$\delta_j = f'(a_j) \sum_k w_{kj} \delta_k. \tag{2.5}$$

The interpretation of these equations is simple. If w_{ji} is the weight of the connection between the neuron j we are considering and a neuron i in a previous layer, then the sum over k relates to all the neurons in the *next* layer to which j connects: this way, since the update of w_{ji}, according to 2.3, is given by

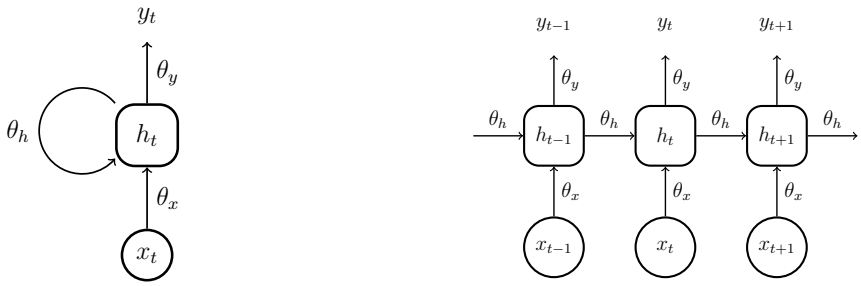
$$w_{ji}^{(\tau+1)} = w_{ji}^{(\tau)} - \eta \frac{\partial E}{\partial w_{ji}} \tag{2.6}$$

we see that, from 2.4 it simply is the product of the error of the current neuron, δ_j, with the output of the previous neuron $f(a_i)$. In turn, from 2.5, we see the recurrence relationship between it and the weighted sum of all posterior neurons that connect to it. Hence, the name backpropagation is now clear: we are, in fact, propagating the errors backwards into the neuron of interest, weighted by the corresponding weight, but now *backwards* instead of forward, as before. For the output units, the δ_j is simply the difference between the produced output and the corresponding label for that sample. This two-step process is performed for every data point in out dataset. For a complete proof of the above formulas, see [6, chap. 5.3.1].

2.1.3 Recurrent Neural Networks

A Recurrent Neural Network (RNN) is, essentially, a regular ANN where some neurons (especially in the hidden layer) have *feedback connections* to themselves, i.e. their outputs are fed as inputs. The relevance of this different structure is the possibility to retain *sequence* information about the data. Before, each incoming data point only contributed to the training of the network, but the information about the correlation between themselves and the data points that preceded them did not influence the training step. Temporal relationship is disregarded and each data point considered conditionally independent of any other. This is obviously not necessarily true, and in fact there are many cases where the correlation between data points is high for those closely spaced in time, such as in video signals, audio signals, or other kinds of *temporal sequences* of data. Therefore, the feedback connection of the neuron to himself acts as a kind of *memory element* that takes into account in the present decision, the history of decisions previously taken, and hence the previous data.

Figure 2.4 suggests a possible structure for a neuron of a hidden layer in an RNN, and also an alternate representation, where the structure is unfold through time.



(a) Recurrent Neuron

(b) Recurrent Neuron, unfolded through time

Figure 2.4: In Figure 2.4a, a recurrent neuron is depicted, where the output is fed back to the input, weighted by θ_h. Figure 2.4b, an unfolding through time of that same neuron is performed. The basic idea is that feeding the output back to the input is similar to feeding the output to the input of the neuron *at a previous time step*: this way, we linearize the structure. Considering a training epoch of T samples, corresponds to having T unfolded neurons/layers

The training of an RNN is usually performed using a variation of Backpropagation, called **Backpropagation Through Time** (BPTT), that as the name implies, performs the same back-propagation procedure discussed for the ANNs, but now taking into account the unfolding of the network through a fixed training epoch T like Figure 2.4b. Due to this very fact, this training procedure is memory and performance consuming, and so it will not be used in my final work, but instead a novel approach, the **Simultaneous Perturbation Stochastic Approximation** will be evaluated.

Even though RNNs outperform static ANNs in sequence recognition problems [7], they fail to retain long-term dependencies. Of course that the weight training process is itself a form of memory, but the problem is that the weight update is much slower than the activations [8], and therefore this memory only retains short-term dependencies. This is because of the so-called **Vanishing Gradients Problem** [9, 8], where the error decays exponentially through time, and the impact of previous incoming data points on the training of the weights, and thus the current decision, quickly decreases.

2.1.4 Long Short-Term Memory Networks

To overcome the issue of failing to remember long-term dependencies, Hochreiter and Schmidhüber proposed, in 1997, a novel approach to the RNNs called the *Long-Short Term Network* [4]. This section explains the main idea of this approach (Section 2.1.4.1), and also how it is trained (Section 2.1.4.2), serving as a support for the work of this thesis. Although originally formulated in 1997, its formulation has been incrementally updated in [10] and [11], and the most current version is the one in [1]. One of the inital proposers of LSTM, Prof. Jürgen Schmidhüber, did a survey on the most common variations of the model last year [2], and this will the basis of this short theoretical presentation, as well as the work that will be developed in this thesis.

2.1.4.1 Structure, Operation and Equations

A single LSTM neuron is presented in Figure 2.5. As we can see from the picture, we still have the recurrent connections from the regular RNNs, but now there are multiple entry points that control the flow of information through the network. Although omitted from the picture, all the gates are biased, as is suggested in Equations 2.7. The main components, their role and relevance, are explained as follows

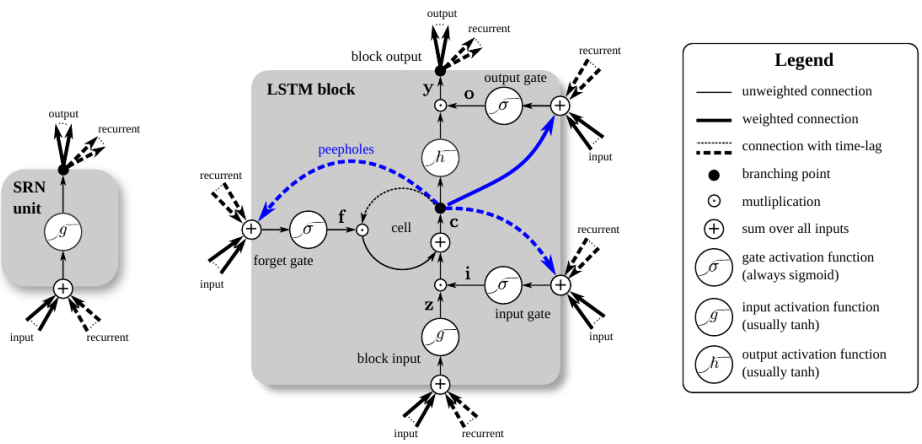


Figure 2.5: A complete LSTM neuron, with all the features as described in [1]. Source: [2]

- **Input Gate** – this is the input gate, where the importance of each feature of the input vector at time t, $\mathbf{x}^t$, and the output vector at the previous time step $\mathbf{y}^{t-1}$ is weighed in, producing an output $\mathbf{i}^t$.

- **Block Input Gate** – as the name implies, this gate controls the flow of information from the input gate to the memory cell. It also receives the input vector and the previous output vector as inputs, but it does not have peephole connections and its dynamics are controlled by a different set of weights. The **activation function of this gate can be either**, but the most common choice is the **Hyperbolic Tangent**.

- **Forget Gate** – its role is to control the contents of the Memory Cell, either to set them or reset them, using the *Hadamard Elementwise* matrix multiplication of its output at time t, $\mathbf{c}^{(t)}$, with the contents of the memory unit at the previous time step, $\mathbf{c}^{(t-1)}$. The activation function of this gate is **always sigmoid**.

- **Output Block Gate** – this gate has a role very similar to that of the Block Input Gate, but now it controls the information flow *out* of the LSTM neuron, namely the activated Memory Cell output.

- **Memory Cell** – the cornerstone of the LSTM neuron. This is the memory element of the neuron, where the previous state is kept, and updated accordingly to the dynamics of the gates that connect to it. Also, this is where the peephole connections come from.

- **Output Activation** – the output of the Memory Cell goes through this activation function that, as the gate activation function, can be any, but the *hyperbolic tangent* is the most common choice.

- **Peepholes** – direct connections *from* the memory cell that allow for gates to 'peep' at the states of the memory cell. They were added after the initial 1997 formulation, and their absence was proven to have a minimal performance impact [2].

After these small conceptual definitions, that allow us to grasp some intuition on the operation of a *single* LSTM cell, I can present the overview of a *layer* of LSTM neurons and their formal mathematical formulation, that will be needed both for the high-level model and the HDL description. The operation of each set of gates of the layer is given by the following set of equations

$$
\begin{aligned}
\mathbf{z}^{(t)} &= g(\mathbf{W}_z\mathbf{x}^{(t)} + \mathbf{R}_z\mathbf{y}^{(t-1)} + \mathbf{b}_z) \\
\mathbf{i}^{(t)} &= \sigma(\mathbf{W}_i\mathbf{x}^{(t)} + \mathbf{R}_i\mathbf{y}^{(t-1)} + \mathbf{p}_i \odot \mathbf{c}^{(t-1)} + \mathbf{b}_i) \\
\mathbf{f}^{(t)} &= \sigma(\mathbf{W}_f\mathbf{x}^{(t)} + \mathbf{R}_f\mathbf{y}^{(t-1)} + \mathbf{p}_f \odot \mathbf{c}^{(t-1)} + \mathbf{b}_f) \\
\mathbf{o}^{(t)} &= \sigma(\mathbf{W}_o\mathbf{x}^{(t)} + \mathbf{R}_o\mathbf{y}^{(t-1)} + \mathbf{p}_o \odot \mathbf{c}^{(t)} + \mathbf{b}_o) \\
\mathbf{c}^{(t)} &= \mathbf{i}^{(t)} \odot \mathbf{z}^{(t)} + \mathbf{f}^{(t)} \odot \mathbf{c}^{(t-1)} \\
\mathbf{y}^{(t)} &= \mathbf{o}^{(t)} \odot h(\mathbf{z}^{(t)})
\end{aligned}
\tag{2.7}
$$

where $\odot$ is the Hadamard multiplication. The i-th element of the previous vectors in bold corresponds to the value of the gate of the i-th neuron of the layer, which is a very convenient and compact representation of the whole layer. Furthermore, if the layer has N LSTM neurons and M inputs (i.e. the size of the layer that precedes this), we see that the input weight matrices $\mathbf{W}_*$ and the recurrent weight matrices $\mathbf{R}_*$ all have size $N \times M$, and that the bias weight matrices $\mathbf{b}_*$, the peephole weight matrices $\mathbf{p}_*$ and the matrices $\mathbf{z}^{(t)}$ through $\mathbf{c}^{(t)}$ have size $N \times 1$. Although useful for a high-level description in a programming language, the matrix representation may not be suitable for a direct HDL port. In that case, in order to represent the operation of the *i-single neuron*, all the above equations still hold, but instead of *vectors* of outputs, we will have a single scalar output, and we only use the appropriate row in the weight matrices $\mathbf{W}_*$, $\mathbf{R}_*$ and the remaining.

2.1.4.2 Training – SPSA

Since this work aims to propose an on-chip learning circuit, it is important to find a suitable learning scheme. Since I am aiming at an hardware implementation, and although the memory resources of current FPGAs are abundant, one must find an algorithm that uses as less memory as possible (at the smallest performance cost possible). That being said, we see that the usual training algorithms for LSTM cells — i.e. Real-Time Recurrent Learning (RTRL), Truncated Backpropagation Through Time (BPTT) or a mixture of both [1] [4] [2] — usually involve the storage of the deltas of each layer for every time instant in the training epoch (from 0 to T), which

is a highly non-scalable solution both in terms of memory and performance. A most efficient approach to training times series dependant structures like LSTM is the use of **Simultaneous Perturbation Stochastic Approximation** (SPSA) [12]. The main idea of this technique is, instead of explicitly evaluating the gradients for the cost function at each time step, to perform a random walk in the neighbourhood of the current weight matrix, in the weight space, and approximate the new weight update by the approximation of the gradient of the cost function that resulted from that random walk. The weight update for the i-th weight at the time step t is given by

$$\Delta w_i^{(t)} = \frac{J(\mathbf{w}^{(t)} + \beta \mathbf{s}^{(t)}) - J(\mathbf{w}^{(t)})}{\beta s_i^{(t)}} \tag{2.8}$$

where β is the magnitude of the perturbation to be introduced, $\mathbf{s}^{(t)}$ is a *sign vector* whose i-th element, $s_i^{(t)}$, is either -1 or 1. This way, we see that every weight is **randomly** incremented either by $-\beta$ or β, and we only need to keep a duplicate of the weight matrix with the perturbations, and we only need to evaluate the cost function twice per incoming sample. As for the *update rule*, we have

$$w_i^{(t+1)} = w_i^{(t)} - \eta \Delta w_i^{(t)} \tag{2.9}$$

where η is the learning rate, and $\Delta w_i^{(t)}$ is the update for the i-th weight evaluated in 2.8. According to the analysis performed in [13], performing a two-term Taylor expansion at $\mathbf{w} = \mathbf{w}^{(t)}$, and taking the expected value of the equation, we get that there is a $\mathbf{w}_{S1}$ so that

$$\Delta w_i^{(t)} = s_i^{(t)} (\mathbf{s}^{(t)})^T \frac{\partial J(\mathbf{w}^{(t)})}{\partial \mathbf{w}} + \frac{c s_i^{(t)}}{2} (\mathbf{s}^{(t)})^T \frac{\partial^2 J(\mathbf{w}^{(t)})}{\partial \mathbf{w}^2} (\mathbf{s}^{(t)}) \tag{2.10}$$

If the random sign vector $\mathbf{s}^{(t)}$ is chosen carefully so that the expected value of the vector at the i-th position is zero, $E(s_i^{(t)})$, and the signs of two different components i and j, at different time-steps, are independent (i.e. $E(s_i^{(t_1)} s_j^{(t_2)}) = \delta_{ij}\delta_{t_1 t_2}$, where δ is the Kronecker function), taking the *Expected Value* of equation 2.10 yields

$$E(\Delta w_i^{(t)}) = \frac{\partial J(\mathbf{w}^{(t)})}{\partial w_i^{(t)}} \tag{2.11}$$

that is, the *expected value* of the weight update approximates the gradient of the cost function relative to that weight, and so the learning rule is a special form of *Stochastic Gradient Descent*.

One last comment to be made concerning the update rule, is the hypothetical need for **limits** to the weight values, when the update rule exceeds $|w_{\max}|$ (in that case, we set $w_i^{(t+1)} = \pm w_{\max}$), which sometimes might be needed if the behaviour of the weight update is not appropriate [13].

Chapter 3

State of the Art

Over the course of this chapter, I am going to present an overview of the most recent developments related to the work of this thesis, both in terms of existing dedicated hardware implementations 3.2, the most relevant work in adapting suitable training algorithms to hardware 3.3 and also some of the most relevant applications of LSTM 3.1, which are not FPGA-based, but demonstrate how LSTM are useful by themselves, and how well it competes with other Machine Learning algorithms in terms of long time-series dependences in data.

3.1 LSTM Applications (non-FPGA related)

LSTM Networks are nowadays one of the state of the art algorithms in deep-learning, and their performance is superior to that of other kinds of RNNs and Hidden Markov Models, both of which are generally used to address the same set of problems where LSTM are employed, namely predicting and producing classification decisions from time-series data. A very comprehensive description of applications can be found in one of the initial authors webpage dedicated to the subject [1]. I will now enumerate some of the leading edge applications of LSTM.

- **Handwriting Recognition** – an LSTM-based network [14], submitted by a team of the Technische Universität München, won the 2009 ICDAR Handwriting Recognition Contest, achieving a recognition rate of up to 91% [15]. LSTMs have also proven to surpass HMM-based models in terms of optical character recognition of printed text, as [16] suggests.

- **Speech Recognition** – an architecture[17] proposed by Graves et al. in 2013 achieved an astonishing 17.7% of accuracy on the TIMT Phoneme Recognition Benchmark, which up to the date is a new record. Furthermore, it has also been used for large scale acoustic modelling of speech [18].

- **Handwriting Synthesis** – a comprehensive study by Graves shows, among other sequence generation tasks such as Text Prediction, that use of LSTM to produce synthetic handwriting, that looks incredibly similar to human-produced handwriting [19].

[1] http://people.idsia.ch/ juergen/rnn.html

- **Translation** – LSTM was used by Sutskever et al. (Google) to perform sequence-to-sequence translation on the WMT'14 dataset, translating English to French with a close to state of the art score [20].

- **Biomedical Applications** – this network architecture was used in a protein homology detection scheme [21] using the SCOP 1.53 benchmark dataset, displaying a good performance when compared to other methods. Similarly, a recent article from 2015 by Sønderby et al. suggested the use of standalone LSTM and also Convolutional LSTM to perform sub cellular localization of proteins, given solely their protein sequence, with an astounding accuracy of 0.902 [22]

- **Music Analysis and Composition** – Lehner et al. proposed a low-latecy solution based on LSTM, suitable for real-time detection of a singing voice in music recordings [23] whose performance surpassed other baseline methods, with lower latency. In terms of music transcription from audio data, there is a study that proposes the use of LSTM cells to perform a transcription of piano recordings to musical notes [24], in order to automate music transcription. The model was trained with recordings of both acoustic pianos and synthesized pianos, and the labelling was performed using an associated MIDI file for each piece that was used in the training, showing promising results. [25] suggests the use of LSTM to perform autonomous computer music composition, and Eck and Schmidhüber proposed LSTM to perform Blues improvisation in [26].

- **Video and Image Analysis** – Vinyals et al., at Google, proposed an LSTM network for **image captioning**, preceded by a Convolutional Neural Network (CNN) to apply preprocessing to the images [27]. The LSTM works as *sentence generator*, that captions the images with state of the art performance. Besides image captioning, video captioning is also an interesting topic. Venugopalan et al. recently proposed a CNN + LSTM architecture to translate video sequences to natural language [28], using the Microsoft Research Video Description Corpus as a dataset.There are other similar studies that combine image and video captioning [29].

3.2 Hardware Implementations of LSTM

Hitherto, there is but one actual implementation of an LSTM network in hardware, published recently (Januray 2016) by Chang et al. [5] in the international Journal of Advanced Research in Electrical, Electronics and Instrumentation Engineering. It consists on the proposal of an LSTM network with $N = 128$ for dedicated hardware, targeting a Xilinx® Zedboard implementation. It uses a character-level language model from Andrej Karpathy [2], written in Lua using the Torch7 framework [3].

[2] https://github.com/karpathy/char-rnn
[3] http://torch.ch/

Although the article is not clear on whether there is active learning by the ARM CPU – the authors refer that the CPU loads the weights before operation, and that it changes them during operation, although how and why that change is done is not even clearly explained, neither mathematically nor conceptually –, **there is no on-chip learning module** in the FPGA according to the description provided. For that reason, the system stores the weights in external memory, and accesses them using Direct Memory Access (DMA). Furthermore, the level of parallelism is low, since for an $N = 128$ network, the design uses only 22.73% of the DSP48's available in the board. This is because the matrix multiplications are serialized, and only one DSP is used for each matrix-vector product (aside from other uses).

The implementation is not flexible, since there is nothing that suggests a parameterization of the HDL description. Hence, it is only possible to synthesize networks of size $N = 128$, meaning that there needs to be a redesign of the network for other network sizes. On the other hand, this is a valid approach for special applications that require networks of this size, and where we only need a co-processor for LSTM networks, and not a dedicated standalone system.

3.3 Training Algorithms and Implementation

As stated in 3.2, the work of [5] does not feature on-chip learning at FPGA level, although there are a handful proposed solutions for it in recent literature. I will particularly look into the ones that use SPSA (see Section 2.1.4.2), since that is the training algorithm of choice for my proposed solution, and also to the ones that particularly apply SPSA to the training of Neural Networks [13].

The SPSA algorithm was initially published by Spall in [12], and its theoretical details, are outlined in Section 2.1.4.2. As to its applications to the training of general Neural Networks, the earliest examples come from 1995 and 1996 in [30, 31] where SPSA is used to train a VLSI *Analog* Neural Networks, a time where the memory resources of digital circuitry were limited, and so most of these structures were analog-based. Its adequacy was also established for **control problems**, such as those proposed in [32], where a Time Delay Neural Network is used to control an unknown plant in a linear feedback system.

In 2005, Maeda and Wakamura published a proposed SPSA hardware implementation [13] to train an Hopfield Neural Network in an FPGA (and thus a digital system), achieving promising results in an Altera FPGA. The article carefully delineates the approach taken, and also the hardware architecture designed.

Furthermore, a 2013 article by Tavear et al. [33] proposes, for the first time, using SPSA to train LSTM Neurons, although the article focuses on proving the suitability of SPSA to LSTM, and no actual hardware implementation is done or proposed. The authors simply demonstrate the suitability using conceptual arguments and by building a software model of an SPSA-trained LSTM network, and by comparing both the performance and computing speed of their model with the results achieved by Hochreiter et al. in [21]. Since the forward phase in both regular LSTM and SPSA-trained LSTM is the same, the computation time suffers no performance penalty whatsoever

and the learning ability is preserved to a high degree, showing that SPSA is a valid alternative do BPTT and other similar and more common training schemes.

3.4 Final Overview

As we could attest from this small literature survey, although there is already an hardware implementation of LSTM, there is still a good deal of room for improvement by adding on chip-learning to the system, and also to improve its performance and flexibility. Furthermore, [33] shows that, at least for that particular case, the LSTM network doesn't suffer a great performance impact from using SPSA training, as opposed to the more common BPTT, and [13] showed that an SPSA hardware implementation is feasible.

Chapter 4

Proposed Architecture

4.1 Constituent Modules

4.1.1 Finite-precision representation system

Before discussing any other details concerned with the actual hardware implementation, some of the design choices that were made regarding how the numbers involved in the calculations are represented are laid out. For that purpose, Section 4.1.1.1 discusses the fundamentals of fixed-point representation systems, as well as the bitwidth and precision chosen for the number representation system of the network, and Section 4.1.1.2 states how the conversion between real and fixed-point numbers is performed. Finally, Section 4.1.1.3 explains the special rules that fixed-point arithmetic imposes. The considerations made in this chapter can be found on [34], which is a good reference for fixed-point arithmetic theory.

4.1.1.1 Precision bitwidth

Since we are dealing with real numbers, and the plan is to make use of the DSP48 slices within the FPGA, an 18-bit signed fixed-point system was chosen, with the sign information coded as 2's complement. Fixed-point systems are usually addressed in the $Qn.m$ form, where n is the bitwidth of the of the integer part (excluding the sign bit) and m is the bitwidth of the fractional part, and so the total bitwidth is $N = m + n + 1$ to account for the sign bit. In this way, the value of the i-th position bit is 2^{i-m}, and since this is a 2's complement system, the last bit is worth -2^{n-1}. In terms of range of representation, the maximum positive number that can be represented corresponds to all bits set to one, except for the last $(2^{n+m+1} - 1)$, but shifted by m bits to the right to yield the correct real number (the decimal point is at the m-th bit), i.e.

$$\text{Max. Positive Number} = \frac{2^{N-1} - 1}{2^m} = 2^{N-1-m} - \frac{1}{2^m} \tag{4.1}$$

and the smallest negative number is simply the MSB set to one (and also shifted appropriately),

$$\text{Small. Negative Number} = -\frac{2^{N-1}}{2^m} = -2^{N-1-m}. \tag{4.2}$$

Assuming that the smallest perturbation used in the training system will be 2^{-9}, according to the previous Python experiments, the fractional part precision should be, at least, greater than this, so a sensible choice would be either $m = 10$ or $m = 11$. On the other hand, given that, in the Python experiments, it was attested that the weight values generally do not (and should not) exceed values around the first power of ten, there is no need for large values of n (although it is advisable to be large enough to accommodate the intermediate calculations, avoiding overflow), and so we should choose to have as much precision as possible, so m was set to 11. Since $18 = n + m + 1$, then $n = 17 - m = 6$ and the representation system to be used is $Q6.11$. According to Equations 4.1 and 4.2, the real values x than can be represented with this choice of n and m are in the range

$$-64 \leq x \leq 63.99951172 \tag{4.3}$$

with a minimum resolution of $2^{-11} = 0.00048828125$.

4.1.1.2 Conversion between real and fixed-point

In order to find the real number equivalent of a $Qn.m$ fixed-point system, and *vice-versa*, we need to take into account that, according to Section 4.1.1.1, the i-th position bit is worth 2^{i-m}, and therefore the decimal point in this fixed-point system is at position $i = m$, since $2^{m-m} = 1$. This way, the rules are as follows

- **Positive Real to** $Qn.m$ – since the 1 is at bit m, we simply multiply the real number by 2^m, and discard the fractional part of the result
- **Negative Real to** $Qn.m$ – we disregard the sign in the real number, and perform the same operation as before, but we then convert the resulting binary number to two's complement, i.e. by performing bitwise negation, followed by summing 1.
- **Positive** $Qn.m$ **to real** – multiply the fixed-point number by 2^{-m}, to shift the decimal point back by m positions.
- **Negative** $Qn.m$ **to real** – convert from two's complement by performing bitwise logic negation, followed by summing one; then scale the decimal point back by m positions by multiplying by 2^{-m}.

4.1.1.3 Fixed-point arithmetic rules

The three main operations needed in my network design are **signed sums**, **signed multiplications** and **arithmetic shifts** (i.e. the ones that preserve the sign of the MSB) to implement divisions/-multiplications by powers of two. In terms of signed sums, the rule is simple: both numbers need to be scaled to the same base, with their m's being same, so that the decimal point is in the same place in both numbers (n, however, can be different, since that only means that one number is longer than the other, and the missing bits in the smaller one are interpreted as zeros).

For signed multiplication, since both operands are in fixed-point $Qn.m$, and are thus scaled by 2^m, we need to scale the result by multiplying it by 2^{-m} (or perform an arithmetic right shift of m bits). This is because, if a and b are the real numbers to be multiplied, and c is the result, we have that, in fixed-point arithmetic, the result is

$$(a \cdot 2^m) \cdot (b \cdot 2^m) = c \cdot 2^{2m}. \tag{4.4}$$

Since in $Qn.m$, all numbers are represented as $r2^m$ with respect to their real counterpart r, we need to scale back the decimal point in the result of Equation 4.4. We can see that it can be easily achieved by dividing by 2^m, as stated in the previous paragraph.

4.1.2 Activation Function Calculator

In order to evaluate the non-linear activation functions $\sigma(x)$ and $\tanh(x)$, since there is no algorithm that can directly compute them, a suitable way to compute them accurately had to be found, using a finite number of elementary operations, multiplications and additions, that can be performed efficiently by specially tailored blocks within the FPGA (DSP48 slices for multiplication, for instance). For that purpose, after using [35] as a reference on elementary function approximation algorithms, **Polynomial Approximations** were used, since evaluating a polynomial does not have high memory usage needs (as opposed to Table Methods, for instance) and, if the polynomial degree is sufficiently low, the number of multiplications needed is low enough to not pose a restriction both on resources (now DSP slices, and not memory) and in speed (number of number of clock cycles needed to output a result).

4.1.2.1 Theoretical considerations on the approximation method

The polynomial approximation methods aim to approximate some function $f(x)$ in an interval $[a,b]$ using a polynomial $p_n^* \in \mathscr{P}$ of degree n, in order to meet the optimization criteria chosen *a priori*. This optimization criteria can be either the well-known **Least Squares Approximation** procedure, where we minimize the *average quadratic error* $[f(x) - p^*(x)]^2$, or the **Least Maximum Approximation**, where we *minimize* the *maximum possible error*, also commonly called a *minimax* approximation. Since we are operating in $Q6.11$ fixed-point arithmetic, we need to guarantee that the maximum approximation error is close to the precision limit of this representation, 2^{-11}, and so a *minimax* approach is desirable, since it guarantees that a given maximum error is not exceeded. Weierstrass's Theorem guarantees that there is always a polynomial that can approximate any continuous function f with error $\varepsilon > 0$. Chebyshev also proved [35] that, in a *minimax* polynomial approximation of degree n, the minimum approximation error ε is achieved in at least $n+2$ points, and the sign of the approximation error alternates from one interval to the

other, meaning that the error is not *biased* and has zero average. This leads to a linear system of $n+2$ equations, whose i-th line is given by

$$p(x_i) - f(x_i) = (-1)^{n+1}\varepsilon \Leftrightarrow p_0 + p_1 x_i + p_2 x_i^2 + \cdots + p_n x_i^n - f(x_i) = (-1)^{n+1}\varepsilon \tag{4.5}$$

The optimal coefficients of this *minimax* polynomial are found using the *Remez Algorithm*, which provides an iterative approach to solve the linear system given by Equation 4.5 by finding, in each iteration, the $n+2$ set of points x_i of Chebyshev's Theorem that minimize the error function to ε. The algorithm operations are as follows

1. Initializing the set x_i of points to $x_i = \frac{a+b}{2} + \frac{(b-a)}{2}\cos\left(\frac{i\pi}{n+1}\right), 0 \leq i \leq n+1$

2. Solve the system in 4.5

3. Given the polynomial coefficients yielded by step 2, compute the y_i points that minimize $p(x) - f(x)$, and replace the x_is of the next iteration by these y_is. Go to step 2 until ε does not decrease.

On one hand, the system of 4.5 of Step 2 can be written in matrix notation as

$$\begin{bmatrix} 1 & x_0 & x_0^2 & \cdots & x_0^n & -1 \\ 1 & x_1 & x_1^2 & \cdots & x_1^n & +1 \\ & & & \vdots & & \\ 1 & x_{n+1} & x_{n+1}^2 & \cdots & x_{n+1}^n & (-1)^{n+1} \end{bmatrix} \begin{bmatrix} p_0 \\ p_1 \\ p_2 \\ \vdots \\ p_n \\ \varepsilon \end{bmatrix} = \begin{bmatrix} f(x_0) \\ f(x_1) \\ \vdots \\ f(x_{n+1}) \end{bmatrix} \tag{4.6}$$

and therefore the solution vector $p = [p_0\ p_1\ p_2\ \cdots\ p_n\ \varepsilon]$ for Step 2 of Remez's Algorithm is simply given by $p = A^{-1}b$, where A is the matrix on the left-hand side of the equation and b is the vector on the right-hand side. The Python implementation of this algorithm is presented in Listing A.1 of Appendix A.

Instead of using a *single* polynomial of higher degree ($n \geq 3$) for the whole domain, the domain of the activation functions was partitioned in **6 intervals**, and polynomials of degree $n = 2$ were used to approximate the function in each of those intervals. This proved to yield a lower overall approximation error, as expected (the interval on which to perform the approximation is smaller). Also, since both $\sigma(x)$ and $\tanh(x)$ have horizontal asymptotes in $\{0,1\}$ and $\{-1,1\}$ respectively, the far-left and far-right intervals do not need a polynomial approximation, and can be assigned

a constant value equal to the corresponding asymptote. The resulting *minimax* approximation polynomials yielded by running the code in Listing A.1 are

$$\sigma(x) \approx \begin{cases} 0 & x \leq -6 \\ 0.20323428 + 0.0717631x + 0.00642858x^2 & -6 \leq x \leq -3 \\ 0.50195831 + 0.27269294x + 0.04059181x^2 & -3 \leq x \leq 0 \\ 0.49805785 + 0.27266221x - 0.04058115x^2 & 0 \leq x \leq 3 \\ 0.7967568 + 0.07175359x - 0.00642671x^2 & 3 \leq x \leq 6 \\ 1 & x > 6 \end{cases} \tag{4.7}$$

for the sigmoid function and

$$\tanh(x) \approx \begin{cases} -1 & x \leq -3 \\ -0.39814608 + 0.46527859x + 0.09007576x^2 & -3 \leq x \leq -1 \\ 0.0031444 + 1.08381219x + 0.31592922x^2 & -1 \leq x \leq 0 \\ -0.00349517 + 1.08538355x - 0.31676793x^2 & 0 \leq x \leq 1 \\ 0.39878032 + 0.46509003x - 0.09013554x^2 & 1 \leq x \leq 3 \\ 1 & x > 3 \end{cases} \tag{4.8}$$

for the hyperbolic tangent function. These constants were converted to *Qn.m* using the rules in 4.1.1.2, and embedded in the HDL model.

4.1.2.2 Hardware Implementation

The hardware module that implements these activation functions is, essentially, a 2nd degree polynomial calculator, where the coefficients of the polynomial are chosen accordingly with the value of the input x. This last functionality is implemented using a simple multiplexer that loads the signals p_0, p_1 and p_2 with the coefficients of Equations 4.7 and 4.8 based on the value of the input operand x. As for the polynomial calculator is concerned, although we could use a full-pipelined evaluator, that would require two DSP slices – to perform the two simultaneous multiplications – but the DSP slices will be heavily used in the matrix-vector calculators, so it is advisable to save them for that purpose. A simpler approach is to note that, according to Horner's Rule, we get

$$p(x) = p_0 + p_1 x + p_2 x^2 = p_0 + x(p_1 + x p_2) \tag{4.9}$$

where we can note that this operation can be divided in a two-step procedure of a simultaneous multiplication of the operand by a constant, and a subsequent addition of another constant: first, by multiplying the operand by p_2 and summing p_1, and then by multiplying the operand by this last result and summing p_0. The block diagram of the hardware implementation of the non-linearity calculator is presented in Figure 4.1. Also, in Figure 4.2, the output of the Verilog module that implements this design is compared with the actual output (Python3's Numpy was used as reference) for both activation functions. The maximum error for the approximation of the $\sigma(x)$ was of 0.001408 and for the $\tanh(x)$ was 0.0121.

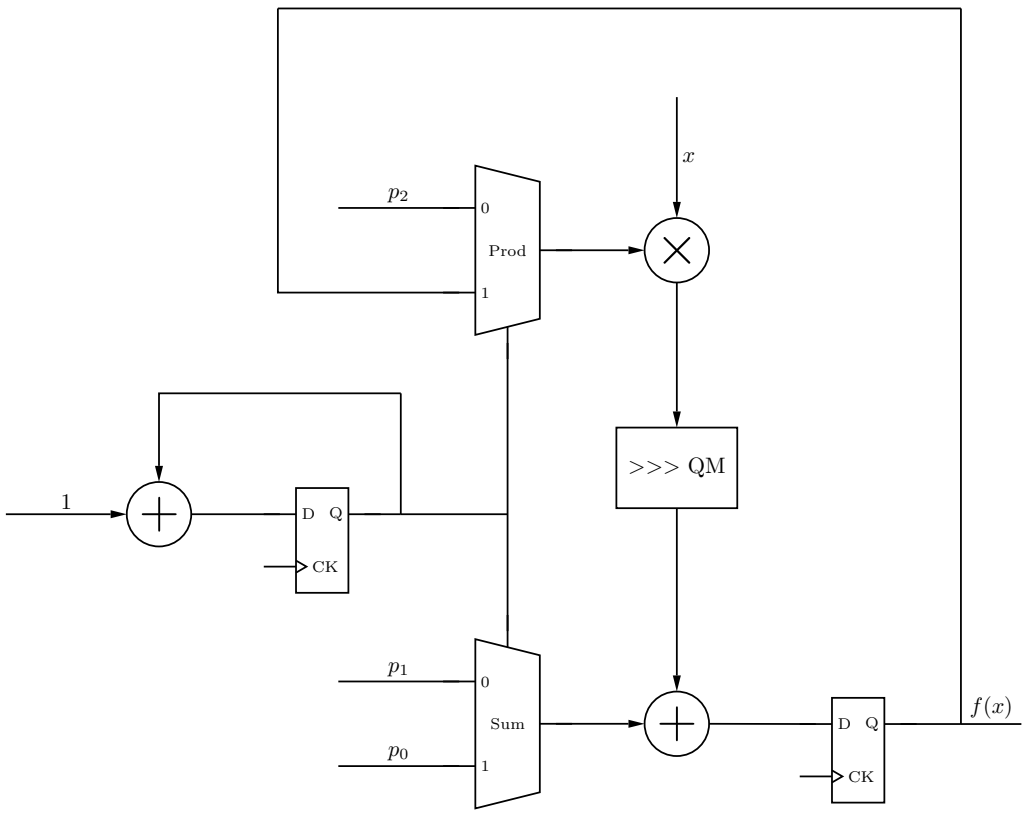
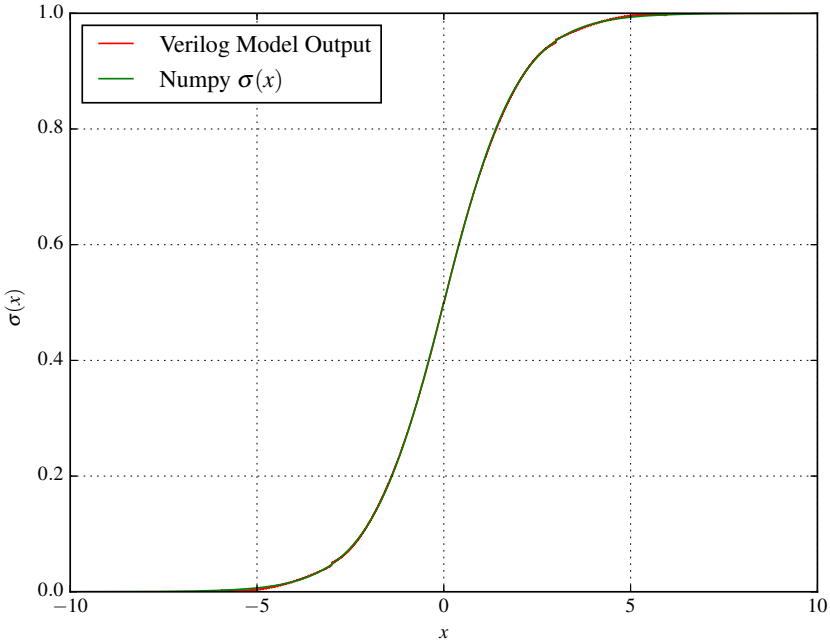
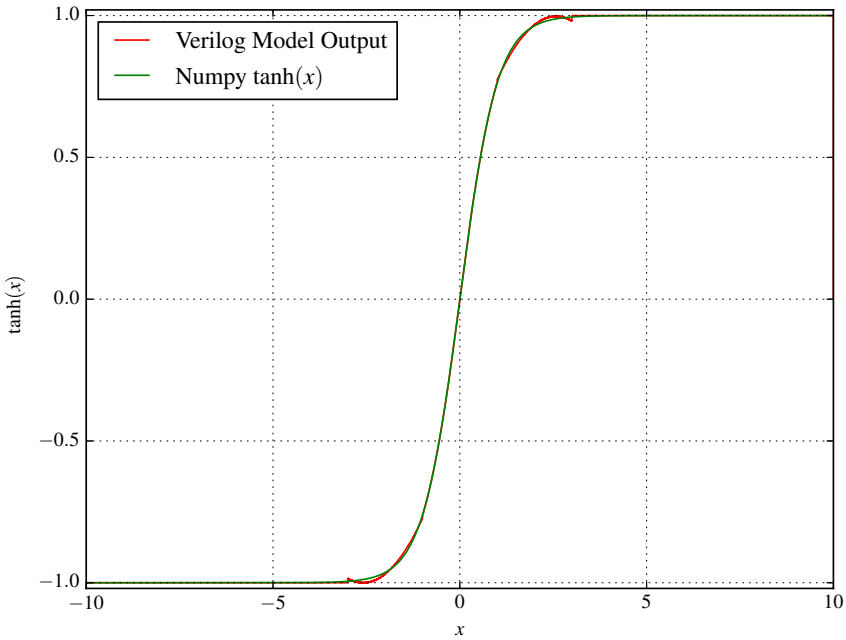


Figure 4.1: Block Diagram of the Activation Function Calculator using a single multiplication. The multiplexer state is controlled by the flip-flip and sum block, that change state every clock cycle. When reset is applied, the selector is set to zero.



(a) Plot of the output of the Sigmoid Calculator HDL module



(b) Plot of the output of the Hyperbolic Tangent HDL module

Figure 4.2: The output of the HDL implementation of the activation functions

4.1.3 Matrix-vector Dot Product Unit

From the set of Equations 2.7, we see that the weight matrices $\mathbf{W}_*$ and $\mathbf{R}_*$ are multiplied by the input vector $\mathbf{x}$ and the layer output vector $\mathbf{y}$, respectively. This way, we need an HDL block that implements matrix-vector multiplication, and the description of that block must be parameterized in order to accommodate different matrices/vectors of various sizes: note that x has length M (the number of inputs to the layer), and so $\mathbf{W}_*$ has size $N \times M$, while y has length N (the number of neurons in the layer), and so $\mathbf{R}_*$ now has size $N \times N$. Plus, a layer with different parameters is needed, either in terms of number of inputs or number of neurons, it is advisable to only change the respective parameter at the synthesis stage, instead of having to redesign the whole block for that particular size.

The matrix-vector dot product of a matrix A of size $N \times M$ with a vector $\mathbf{x}$ of size M, if performed in a linear non-parallel way, can be described in terms of Algorithm 1

Algorithm 1 Matrix-vector multiplication of a matrix

```
for i = 1 : N do
    for j = 1 : M do
        y_i := y_i + A_ij · x_j
    end for
end for
```

This operation has a computational complexity of $O(n^2)$. It can be seen that each of the i-th components of the output vector $\mathbf{y}$ can be calculated **in parallel**, each only requiring the corresponding i-th line from the matrix. Following this approach, matrix-vector multiplication can now be performed in **linear time**, which is one of the great advantages of custom-tailored hardware solutions.

Although this solution only requires one multiplication per row of the input matrix (i.e. N multiplications), if the row size is large, we may quickly run out of resources in the FPGA as N increases; therefore, some sort of *resource multiplexing* strategy must be used to ensure the flexibility of the solution to accommodate networks of larger dimensions. The solution found for this issue was to *share* the multiplication slice between rows of the matrix: in a direct implementation of Algorithm 1, each multiplication slice was responsible for producing the i-th element of the output vector $\mathbf{y}$ (of size N), therefore the final result for the vector would be ready in M clock cycles (i.e. the number of columns); now, if defining a parameter $K_G = \frac{\text{Number of rows}}{\text{Number of multipliers}}$, the number of rows that share the same multiplier, each multiplier is responsible for producing several i-th elements of the output vector, in consecutive time slots of M clock cycles. Suppose that we have a 8×2 matrix, and that $K_G = 2$; in this way, we would have 4 multipliers, and the output vector elements y_0, y_2, y_4 and y_6 would be ready after $M = 2$ clock cycles, and the remaining – y_1, y_3, y_5 and y_7 – are ready after another two clock cycles, that is $2M = 4$ clock cycles after the calculations began.

Figures-4.3 and 4.4 depict a diagram of the memory access for the Matrix, and the row multiplication units within the module, respectively, where $K_G = 4$ was set, for the same matrix and

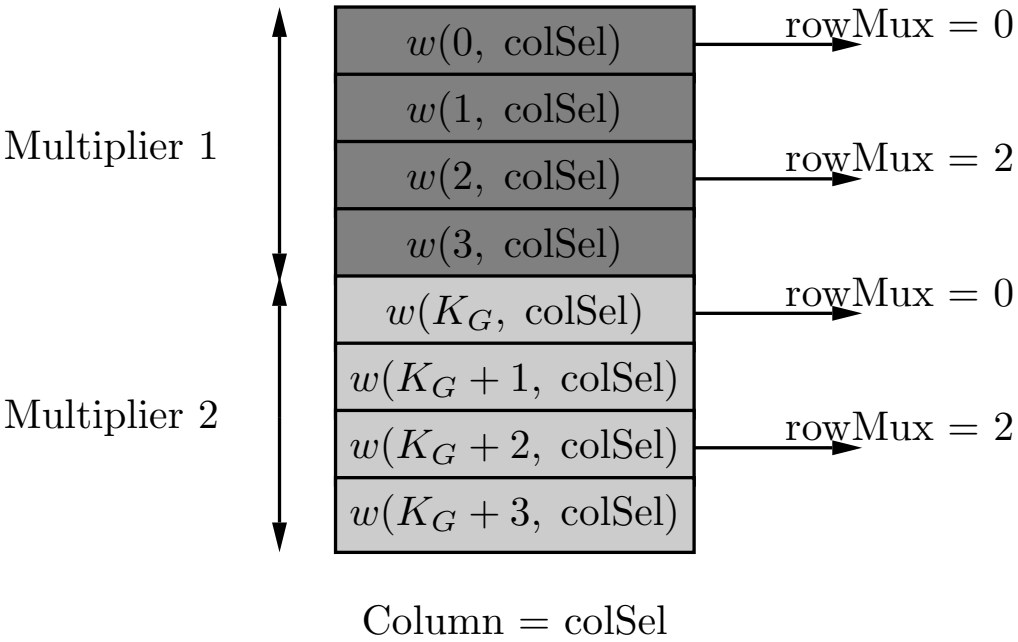


Figure 4.3: A column of the matrix that serves as input to the module. The dark shaded part is for the first multiplier, and the light shaded is for the other, in parallel. The rowMux signal addresses the position within each shaded area

.

vector sizes as before. Note that in this situation there would be only 2 multipliers, and therefore the module would be composed of two multiplication units, such as those in Figure 4.4, that work in parallel: they address a particular column using the signal `colSel`, which is used by the RAM module to output the corresponding column of the matrix (in regard to the input vector, obviously this signal selects only a single position), depicted in Figure 4.3. The dark shaded part of the memory is used by the first multiplier, and the light shaded is used by the other, in parallel for a fixed `rowMux` – this signal is produced by the control unit of the module, and essentially operates the left Mux and right Demux of Figure 4.4 that allows to choose the proper position of the weight column and to write to the correct output register, respectively. In this example, for `rowMux`=0, the control unit increments `colSel` from 0 to M, and thus evaluating y_0 and y_4. After this, `rowMux` is incremented to 1, and the process repeats until `rowMux` reaches $K_G - 1$. Therefore, we have the correct result vector in a time proportional to $K_G \cdot M$: since the memory only outputs the appropriate column in the *next clock cycle*, and that this module is pipelined both at the input and at the output in order to increase the maximum clock frequency, we should add a 3 clock cycle overhead to the previous estimate.

4.1.4 Dual-port weight RAM

Even though bias weight vectors, with size $N \times 1$, were kept as normal registers and the synthesis tool was left to decide how to place them in the FPGA, for the weight matrices the case is different. Since the network size can be large, it is good to make sure that they are placed in the RAMs

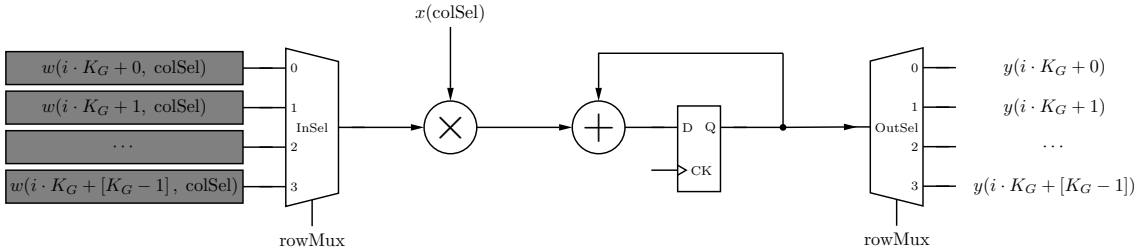


Figure 4.4: The i-th row multiplication unit of the Module, where rowMux and colSel are internal signals produced by the control unit of the module. The flip-flop accumulates the sum, and the output Demux selects the appropriate memory position on where to store this value, within the slot attributed to this multiplication, from $i \cdot K_G$ to $i \cdot K_G + [K_G - 1]$

available in the FPGA. In Section 4.1.3 we see that, because of the architecture of the matrix-vector multiplier, it is convenient to have a column by column access to the weight matrix, and therefore this block was designed to output a given column in the `rowOut` output port terminal, at the *positive edge* of the clock, selected by the value present in the input `addressOut` at the immediately previous *positive edge* of the clock. As far as **writing** to the memory is concerned, the process is identical but now the column of weights at the input `rowIn` is sampled at *positive edge* of the clock, and is placed at the column specified by the input `addressIn` at the previous *positive edge* of the clock.

Note that, for an $N \times M$ matrix, since the memory outputs and writes one column at a time, both the input and output port terminals will carry N weights, and a total bitwidth of `BITWIDTH` $\cdot N$.

The Verilog coding followed the Verilog Coding Guidelines in Xilinx's UG901 Document that, in page 50, recommends that the `RAM_STYLE` parameter be set to `block`. This can be done by adding the following compiler directive before the register definition, as follows

```
(* ram_style = "block" *) reg [PORT_BITWIDTH-1:0] RAM_matrix [NCOL-1:0];
```

where we can see that each register contains the respective column of the matrix, with `PORT_BITWIDTH` $=$ `BITWDITH` $\cdot M$ and `NCOL` $= M$.

After performing synthesis, it was noticed that using block RAM led to a slightly poorer speed performance, so I have settled with the use of *Distributed RAM*, storing the weight matrices in LUTRAM.

4.1.5 Gate Module

The Gate modules are responsible for producing the internal signal vectors for $\mathbf{z}^{(t)}$, $\mathbf{i}^{(t)}$, $\mathbf{f}^{(t)}$ and $\mathbf{o}^{(t)}$. If we note that, according to [2], the removal of peepholes (the signals $\mathbf{p}_*$) does not compromise significantly the performance of the network 2.1.4.1, they were omitted in order to simplify the Gate Module and reduce the usage of DSP slices. This way, a Gate module needs to perform three tasks

1. Multiply matrix $\mathbf{W}_*$ by the input vector $\mathbf{x}^{(t)}$

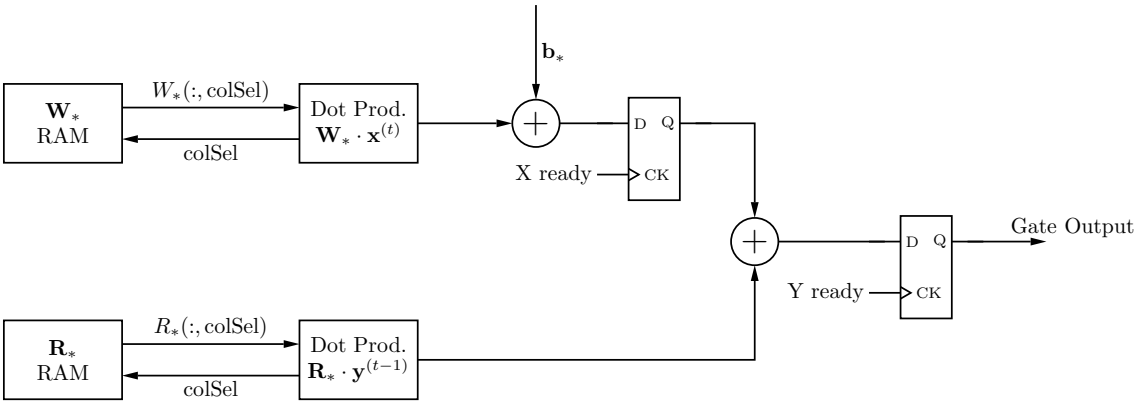


Figure 4.5: Diagram of the hardware block that implements the Gate

2. Multiply matrix $\mathbf{R}_*$ by the previous layer output vector $\mathbf{y}^{(t-1)}$

3. Sum the bias vector $\mathbf{b}_*$ to the remaining matrix-vector dot product results.

Assuming that the network size N is always larger than the input size M, if we use the matrix-vector dot product units of Section 4.1.3, the multiplication in task 1 takes approximately $K_G \cdot M$ cycles and the one in task 2 takes $K_G \cdot N$ cycles. This way, task 2 and task 1 can be performed in parallel, and we can use the extra time that task 2 takes, relative to task 1, to perform task 3, and sum the bias vector to the output of task 1, whose result is ready by that time. The module is triggered by a `beginCalc` input signal that activates the internal state machine, and outputs a `dataReady` signal that informs the network that the calculations have been concluded, and the value at the output is the actual final result. After validating and simulating this block, and taking into account the internal state-machine and that the internal datapath is pipelined, the exact number of clock cycles this module takes to output a result is $6 + K_G \cdot N$.

4.2 Hardware LSTM Network – Forward Propagation

After detailing the design of each of the constituent modules of the LSTM Network, it will now be shown how they are interconnected to implement a hardware version of Equations 2.7. This is performed in Section 4.2.1, where the design decisions and compromises that were taken are detailed, along with some estimates on the resource usage and calculation time for each of the design iteration versions, namely the *naïve solution* (4.2.1.1) and an *optimized version* of the former (4.2.1.2) that exploits some of the hardware redundancies by multiplexing in time the usage of those redundant structures.

4.2.1 Network Structure

Looking at Equations 2.7, we note that the signals $\mathbf{z}^{(t)}$, $\mathbf{i}^{(t)}$, $\mathbf{f}^{(t)}$ and $\mathbf{o}^{(t)}$ do not depend on each other, because they operate only on the current input vector $\mathbf{x}^{(t)}$ and the previous layer output $\mathbf{y}^{(t-1)}$. This way, they can be calculated in parallel, meaning that we need four Gate Modules (see

Section 4.1.5 working in parallel, each one with its respective two RAMs for $\mathbf{W}_*$ and $\mathbf{R}_*$, and followed by the respective activation function calculator (detailed in Section 4.1.2), **one for each** of the N vector elements. There are three elementwise multiplications, two for producing signal $\mathbf{c}^{(t)}$ (which can be done in parallel and then summed elementwise) and one for $\mathbf{y}^{(t)}$ (which can be done only after applying the activation function $\mathbf{c}^{(t)}$).

4.2.1.1 Fully Parallel

By implementing directly the ideas outlined previously, we get the network of Figure 4.6. The memory element of the network is the array of flip-flops that keep the value of the vector $\mathbf{c}^{(t)}$, which is activated every time we have a new incoming signal in order to store the last value, which is now $\mathbf{c}^{(t)}$

In terms of DSP slice usage, this proposed design comprises 3 elementwise multiplications and 5 activation function Modules. Since the number of elementwise multiplications is equal to the network size N, we need $3N$ DSP slices for each elementwise block. As far as the activation function module is concerned, since we need to apply it to each one of the N elements of the Gate's output vectors, and each module uses one multiplication, the ensemble of all 5 modules needs $5N$ DSP slices. Therefore, and noting that each Gate has $\frac{2N}{K_G}$ DSP slices, the total number of DSP slices used is

$$4\frac{2N}{K_G} + 5N + 3N = N\left(\frac{8}{K_G} + 8\right). \tag{4.10}$$

In terms of time performance, we can measure it by estimating the number of clock cycles needed to perform a complete forward propagation. Since the gate module outputs a result in $K_G \cdot N + 7$ cycles, the activation function evaluators need always 5, and the elementwise multipliers need only 2 cycles (one cycle for the pipeline, and another for the actual calculation), and noting that the two elementwise multipliers that sum to produce $\mathbf{c}^{(t)}$ can work in parallel, the estimated number of clock cycles needed would be proportional to

$$(N \cdot K_G + 6) + (5+2) + (5+2) = 20 + N \cdot K_G \tag{4.11}$$

4.2.1.2 Shared Elementwise Multiplication Block

In Figure 4.6, we see that, apart from the elementwise multiplier preceding the register, all of them follow a similar structure: one of the operands is the output of a $\tanh(\mathbf{x})$ block and the other from a $\sigma(\mathbf{x})$. An improvement over the last network architecture is to, instead of replicating these 'tanh-σ-$(\cdot)$wise' structures, use a *single one* and choose the input operands accordingly to the state that the network is currently in. Besides, the right elementwise multiplier of Figure 4.6 is not used as the same time as any other, so it is a waste of resources. The issue about the elementwise multiplier that precedes the register can be solved by adding another multiplexer that chooses between the

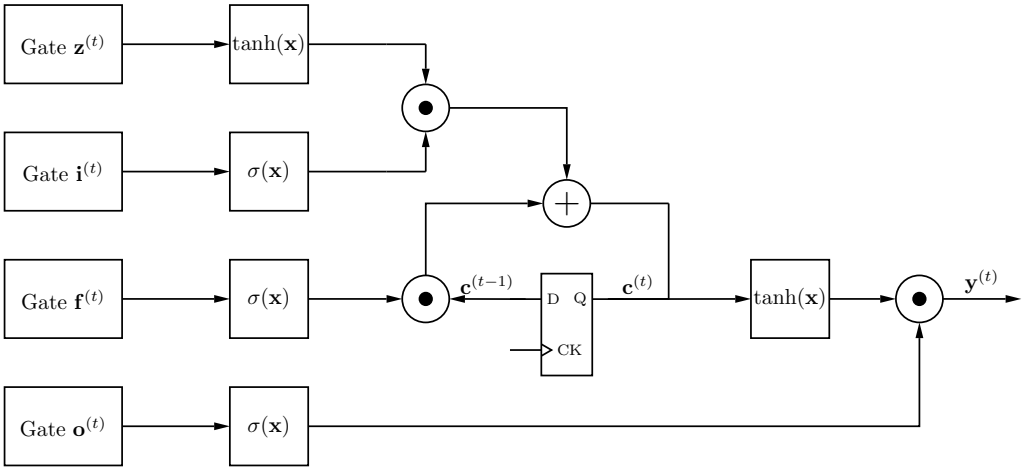


Figure 4.6: Hardware LSTM Network with full level of parallelism

output of the $\tanh(\mathbf{x})$ module or the signal $\mathbf{c}^{(t-1)}$. These ideas resulted in the improved network design of Figure 4.7.

The two left multiplexers control the operands that are fed to the activation function modules, and the selecting signal is generated by the network's state machine and its value is incremented after each complete usage of the 'tanh-σ-$(\cdot)$wise' structure: this is where the time multiplexing of the structure takes place. Since in state `Sel` $= 1$ the left operand of the elementwise multiplier (the one that preceded the register in the previous design) is the signal $\mathbf{c}^{(t-1)}$, another multiplexer was added before the elementwise multiplication, that selects that signal in that particular case, and the output from the $\tanh(\mathbf{x})$ block otherwise.

The registers on the right hand side of Figure 4.7 are activated by signals generated within the network state machine that enable the appropriate register, placing the output from the elementwise multiplicator in the correct place. The first activated register is the middle one, which keeps the result from the operation of the $\mathbf{z}^{(t)}$ and $\mathbf{i}^{(t)}$ vectors, then, after a full operation of the elementwise multiplier, the bottom register is enabled to save the other portion of the sum that evaluates to the $\mathbf{c}^{(t)}$ signal. Lastly, the top register saves the network output $\mathbf{y}^{(t)}$, which in the next incoming

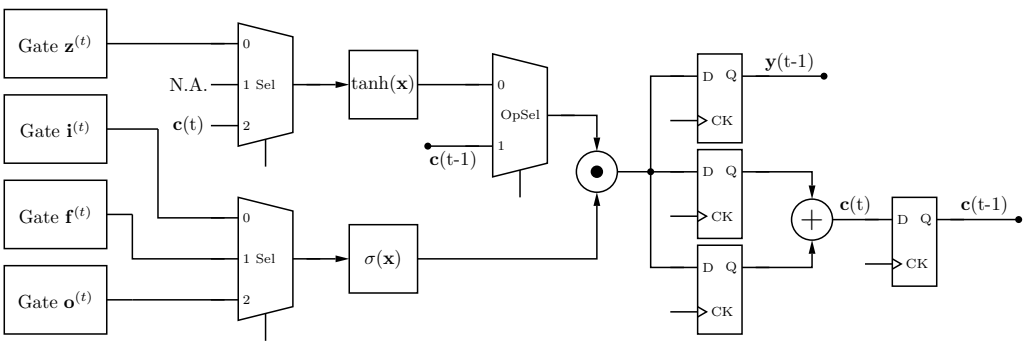


Figure 4.7: Optimised version of 4.6 that shares the elementwise multiplication and the activation function blocks

sample becomes $\mathbf{y}^{(t-1)}$ and is used by the Gate modules in this next batch of calculations.

Now, since there is only a single elementwise multiplier and only two activation function calculators, the total requirement for DSP slices is simply

$$4\frac{2N}{K_G}+2N+N=N\left(\frac{8}{K_G}+3\right). \tag{4.12}$$

where we see that we saved $5N$ multipliers when comparing to the previous design, which for a large value of N can have a decisive impact. In terms of speed performance, although the Gate calculation time remains the same, now the 'tanh-σ-$(\cdot)$wise' structure runs for 3 consecutive times, in a non parallel fashion. After adjustments to the state machine, and accounting for pipelining and synchronization within the datapath, the clock cycles needed after the gate calculations are 27, so the total clock cycles needed are

$$(N\cdot K_G+6)+27=33+N\cdot K_G \tag{4.13}$$

which is only 13 clock cycles more than the fully-parallel architecture. For instance, an $N=32$ neuron network using the fully-parallel design would require 320 DSP slices, while this new architecture would only require 160. Only at the expense of 13 clock cycle overhead.

4.3 Hardware LSTM Network – SPSA Training

In this section, a digital circuit is proposed that aims to train the Network whose details were explained in the previous Section by implementing the training algorithm of Section 2.1.4.2. The training consists in changing the weight values in the RAMs accordingly with the error of the current prediction against the correct outcome. This involves **two** forward propagations: one, where we run the network with the weights unchanged, and another where we *perturbate* the weights by a parameter $\pm\beta$ (the actual sign for each weight is generated at the beginning of the first forward propagation). After these two forward propagations, there is a small time frame where the weights are updated, row by row, before the next incoming sample. This is explained in Section 4.3.2. The way how the numbers are randomly generated in hardware is presented in Section 4.3.1.

4.3.1 Pseudo-random Number Generator

While a True Random Number Generator (TRNG) generates data from a physical entropy source, a Pseudo-random Number Generator (PRNG) generates a sequence of numbers by applying a given function, consecutively, to an initial seed. This means that, if one knows that initial seed and the function that generates the sequence, one can predict the previous and future numbers given the current state. While this is not acceptable for high security cryptographic systems, it serves the purpose of providing a sequence of random sign vectors (from whom both the signal of the perturbation and the update will be chosen from). The remaining issue is to find a PRNG

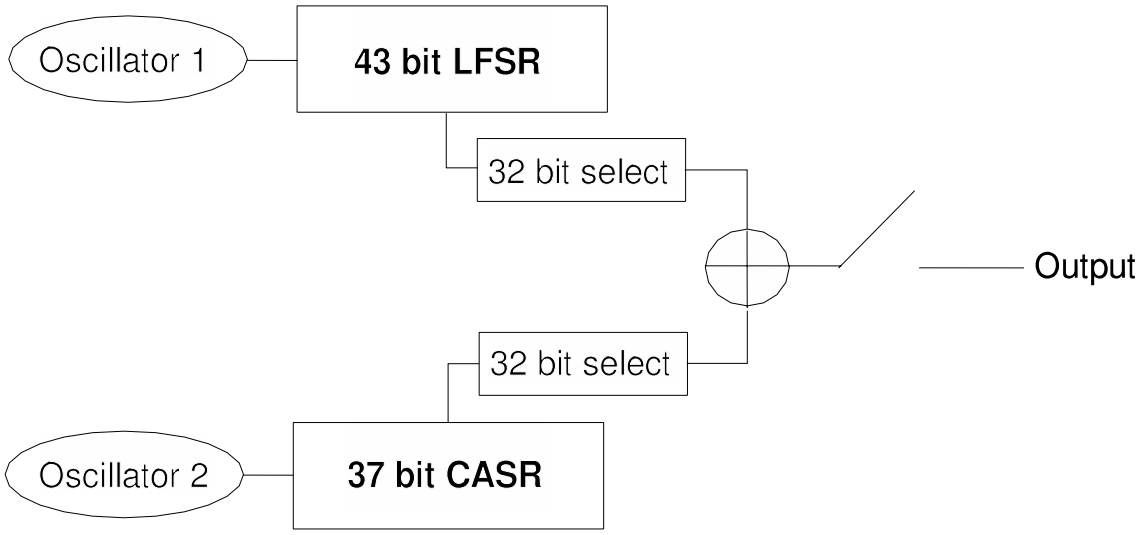


Figure 4.8: The 32 bit Pseudo-random Number Generator. Source [3]

with good statistical qualities, so instead of implementing a simple linear-feedback shift register (LFSR), I did some research on the topic, and eventually chose the solution of [3], based on a 43-bit LFSR and a 37-bit Cellular Automata, by selecting 32 bits from each, permuting the bits and performing a bitwise XOR – the output has, also, 32 bits. The quality of the statistical properties are guaranteed, since it passed the DIEHARD test battery, as well as others, according to [3].

The generator polynomial of the LFSR is a maximal length polynomial, $x^{43}+x^{41}+x^{20}+x+1$, and thus has a period equal to $2^{43}-1$. The CA is also a register, but each bit is updated according to a given **rule** based on the values of the neighboring bits. This particular CA follows Rule 150 for the bit in position $i=27$, given by $c_{t+1}(i)=c_t(i-1)+c_t(i)+c_t(i+1)$, and the remaining bits are updated by Rule 90, given by $c_{t+1}(i)=c_t(i-1)+c_t(i+1)$. The bits positioned at the borders also follow rule 90, but treat as zeros the non-existing i positions.

The output of the hardware implementation of this PRNG is presented in Figure 4.9. It is a state-time diagram, where the contents of the registers at each time are represented by black (ones) and white (zeros). Tthe left column represents the output of the LFSR, the center column is the output of the Cellular Automata and the right column is their combined output. As one can see, the time dependence of the LFSR is very high, which is not desirable. The Cellular Automata, although uncorrelated, still retains some of the patterns that emerge from CAs. On the other hand, the combined output is the one that most resembles white noise, which is a desirable characteristic.

4.3.2 Training Algorithm Circuit

Here is presented the proposed digital circuit of the SPSA training for the network. The circuit of Figure 4.10 is replicated for each of the eight weight RAMs, and the Sign RAM holds a single bit for each of the weight positions of the weight RAMs. This bit keeps the sign of the perturbation to be applied to that particular weight, both on the second forward propagation and on the training stage. During the first forward propagation, and since the weights are not perturbed in this run,

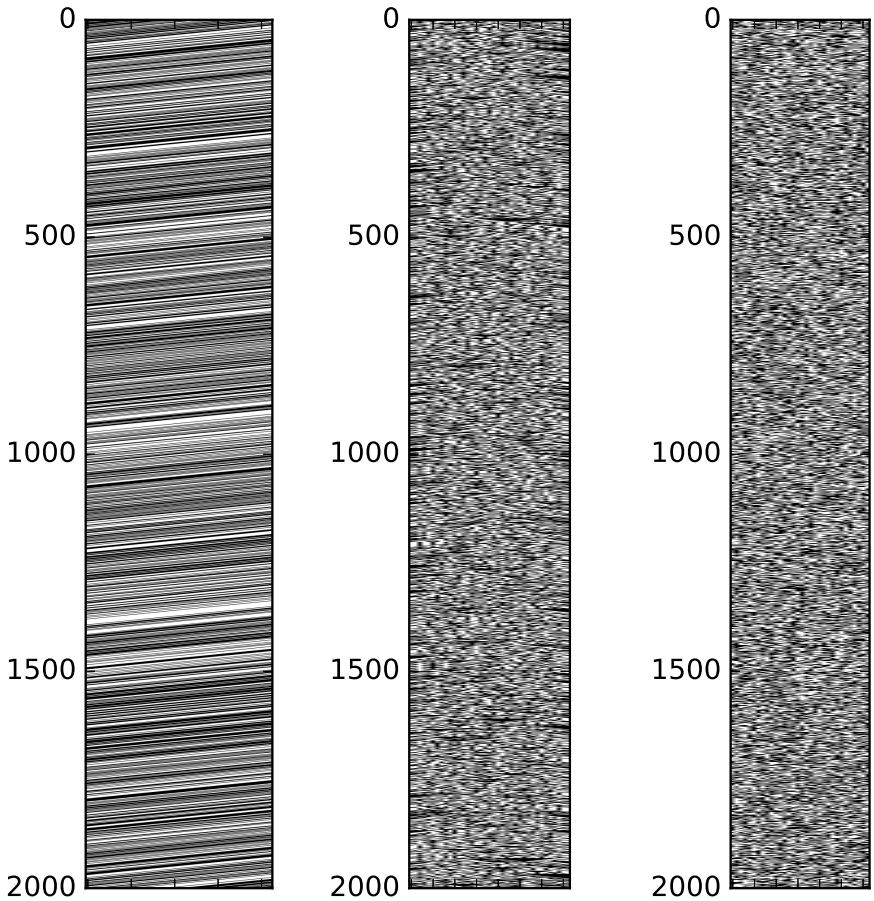


Figure 4.9: The output of the implementation of the presented PRNG (right). The LFSR is the left column, and the CA is at the center. The zeros are represented by white, and the ones by black

we can use this time to populate the sign RAMs with new values for the next training stage, coming from the PRNG of Section 4.3.1. For this purpose, the PRNG is activated by the signal `genRandNum`, generated by the state machine, and we write columns of bits to the sign RAM, re-using the addressing signal produced by the Gate, namely `colAddressRead` (`weightUpdate` is set to zero at this stage). At the second forward propagation, `pertWeights` is set to one, and the Perturbation Mux selects the input where the weights are summed with the perturbation β, whose sign is selected by the sign RAM.

After this stage is completed, and before the network starts a new training cycle for the new incoming sample, we perform the actual training. For that purpose, `weightUpdate` is set to one, and we enable writing to the Weight RAM. Also, the machine state is now responsible for addressing both RAMs, and we begin a simultaneous *pipelined* write to the Weight RAM: this is done by delaying the Write address in respect to the Read address by one clock cycle. This way, at the first clock cycle, while we read Column 0, the next clock cycle will read Column 1 *and* write the sign-changed column 0 plus the update coming from the difference between the outputs in the two forward propagations. In order to simplify the update rule of Equation 2.9, α and β were fixed to negative powers of two, and this replaces the multiplication and division by a simple arithmetic right shift – this idea is also proposed by [13] – considerably simplifying the design.

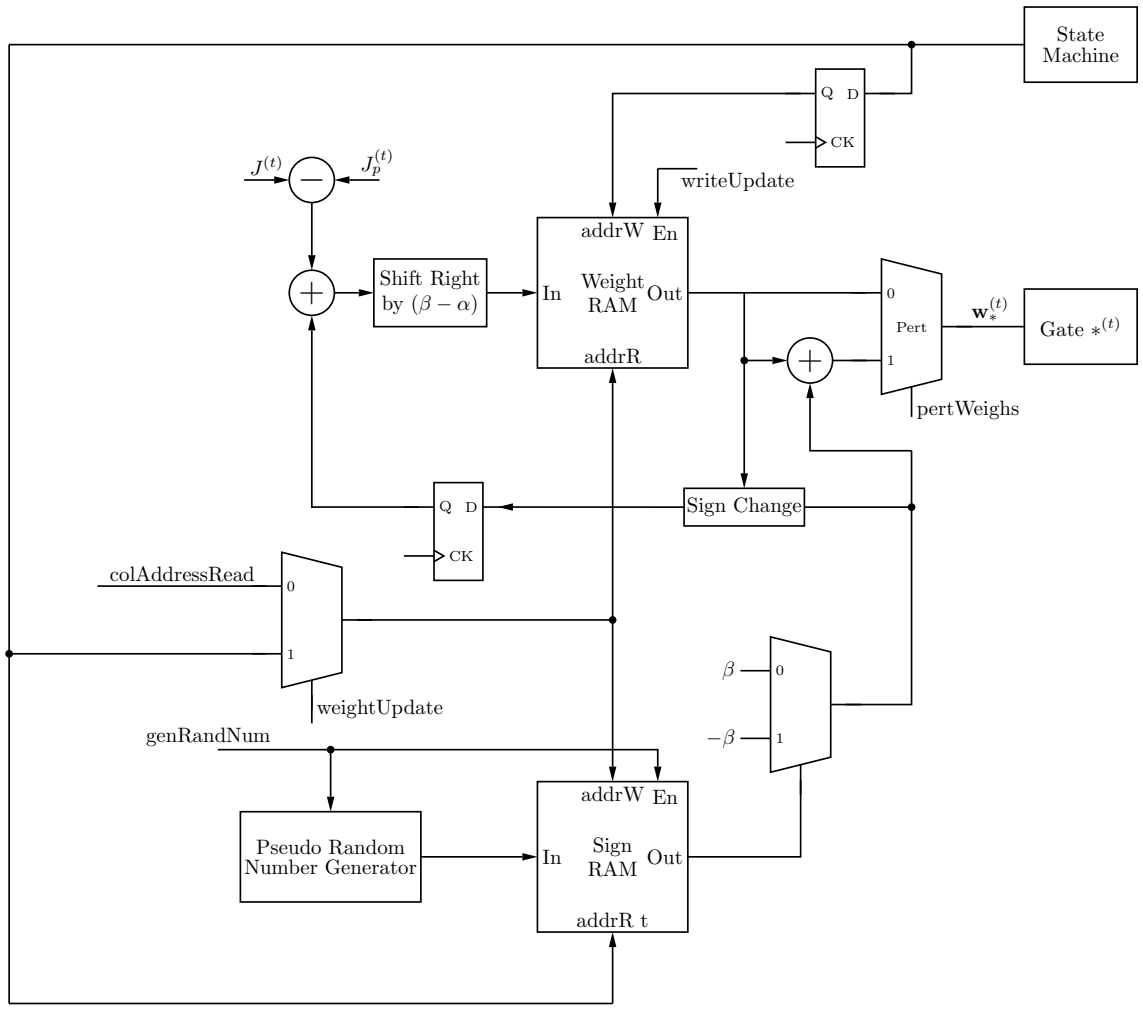


Figure 4.10: Proposed Hardware Implementation of SPSA Training

Chapter 5

Results

The optimized design of Section 4.2.1.2 was synthesized for different network parameters, using Xilinx's Vivado 2015.4 Design Suite, and the synthesis results are reported in Section 5.1, along with their discussion. The performance of the synthesized networks is very promising, showing an approximate 251× speed-up over the LSTM Python model, which was running on a high-end desktop computer with an overclocked CPU, and a 14× speed-up over the most recent hardware implementation [5] Although the FPGA aims for an Embedded System environment usage, the fairness of the comparison with a Desktop computer still holds, since a Python/Numpy implementation is slower than a lower-level language implementation (such as in C) and this level of performance would be similar to a C implementation in a low-power, embedded CPU platform. These results are reported in Section 5.2.

5.1 Synthesis

The proposed network was first synthesized aiming a Xilinx XC7Z020 SoC, for sizes N of 4, 8, 16 and 32, varying the resource sharing parameter K_G, and the number of inputs set to $M = 2$. For a network size of 32 and $K_G = 8$, the LUT usage exceeded the LUT resources available in the FPGA, so only lower values of K_G were successfully synthesized. This is because of the complexity of the input multiplexer of the Matrix-vector Dot Product Module in 4.1.3. To synthesize the design for sizes N of 64, 128 and 256, a Virtex-7 VC707 board was used.

In order to assess the impact of the network size on the performance and resource usage of the proposed design, the Verilog description of the proposed network was first synthesized aiming a Xilinx XC7Z020 SoC platform, for sizes $N \in \{4, 8, 16, 32\}$, and for each of these the resource sharing parameter K_G was also varied (as will be explained further ahead, the number of inputs is set to $M = 2$). Since performance is the key factor for using an FPGA implementation, the synthesis strategy used was the `Flow_PerfOptimized_high`, and the implementation strategy was `Performance_ExplorePostRoutePhysOpt` to perform a post-route optimization stage that proved useful in many cases to meet the clock specifications when the first Place&Route pass failed to meet them. For a network size of 32 and $K_G = 8$, the LUT usage exceeded the LUT

resources available in the FPGA, so only lower values of K_G were successfully synthesized (N.D. in the tables). Networks of sizes $N \in \{64, 128\}$ were also synthesized for a Virtex-7 VC707 board.

5.1.1 Maximum Frequency

Using the synthesis strategy outlined in the previous paragraph, the maximum clock frequency that does not cause timing violations of any sort is plotted in Figure 5.1. For $N = 4$, since there are only 4 rows to be multiplied, the maximum value of K_G is 4, and hence no synthesis was performed for $K_G = 8$ (N.A. in the tables); also, since $K_G = 2$ and $N = 32$ would use $32(8/2+3) = 224$ DSP slices, that exceeds the 220 slices available in the Zynq 7020, so there is no synthesis data for that value.

First, for all network sizes, it is clear that with increasing K_G, the maximum clock speed decreases. This means that there is a critical path in the Matrix-Vector multiplication unit, whose multiplexer becomes increasingly complex for higher values of K_G. On the other hand, when K_G is the same, smaller networks are faster than larger networks. The fastest design is an $N = 4$ and $K_G = 2$ network, with a clock frequency of 158.228 MHz, and the slowest one is an $N = 32$ and $K_G = 4$ network, clocked at 101.523 MHz. The reference design used for validation in Section 5.2 is an $N = 8$ and $K_G = 2$ network is clocked at 154.321 MHz, which yields a clock period of 6.48 ns.

5.1.2 DSP Slices Usage

The estimates made in Equation 4.12 were proven to be accurate, as Figure 5.2 confirms. The reference network design uses 56 DSP slices, which corresponds to 25.45% of the total number of DSP slices available.

5.1.3 Power Consumption

Another important metric of the performance of a design is its **power consumption**. All designs yielded a constant baseline value for the *static* power consumption of around 120 mW, and the power consumption reported in Figure 5.3 refers to the *total* power consumptions, i.e. both the baseline static power and the dynamic power consumption. It is clear that the smaller the network is, the less power is consumed, as one would expect. Furthermore, an increasing level of resource sharing, K_G, yields a substantially lower power consumption figure: this is because less DSP slices are used when K_G is increased. Of course, even though the power consumption is lower in that case, that comes at the expense of a lower clock frequency and a higher computation overhead per forward propagation.

5.1.4 Other Resources Usage

The LUT, LUTRAM and Flip-Flop usages are discussed here. In Table 5.1, the usage of LUTs is reported, where we can see that although there is not a clear trend on how the LUT usage varies

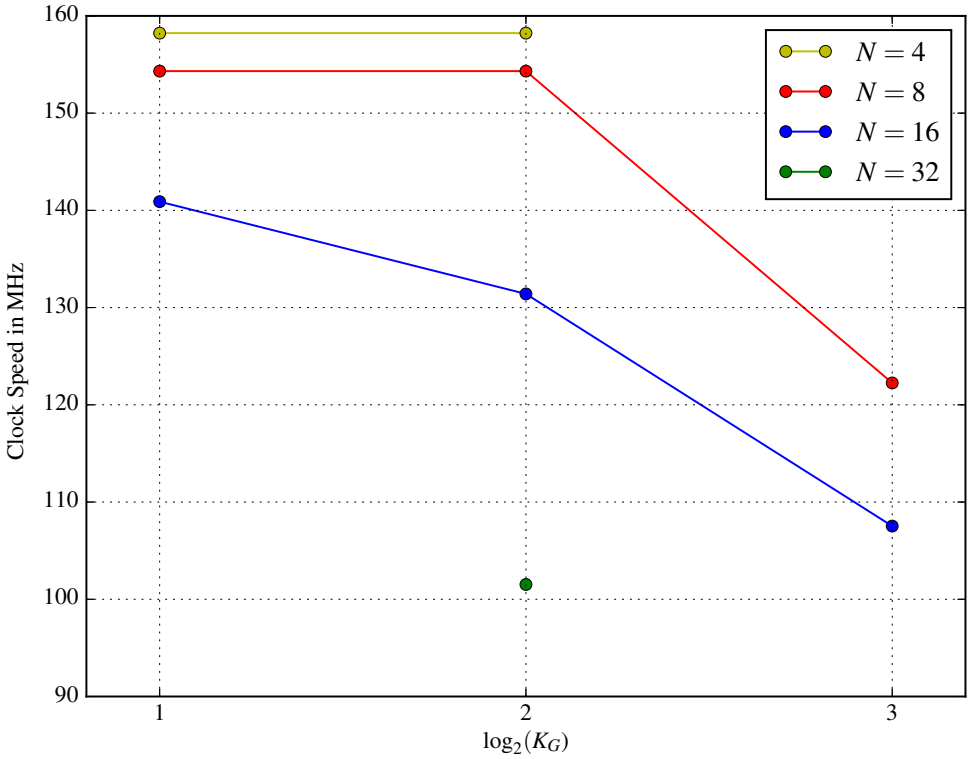


Figure 5.1: The maximum achievable clock frequency for several Network Sizes N and resource sharing level K_G

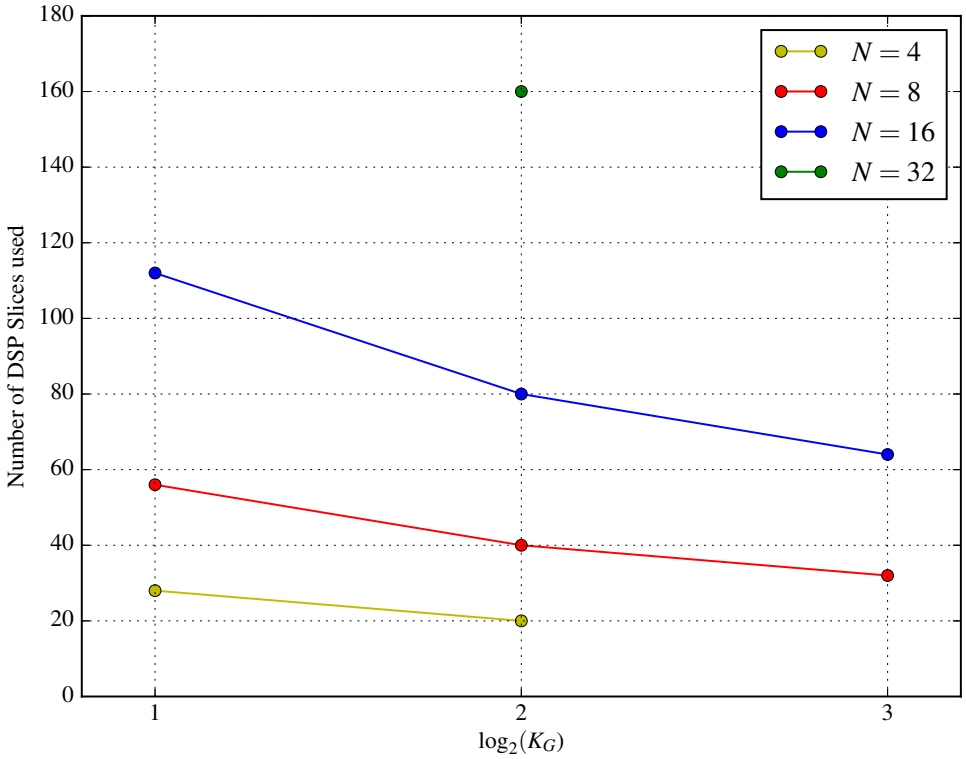


Figure 5.2: The number of DSP slices used for several Network Sizes N and resource sharing level K_G

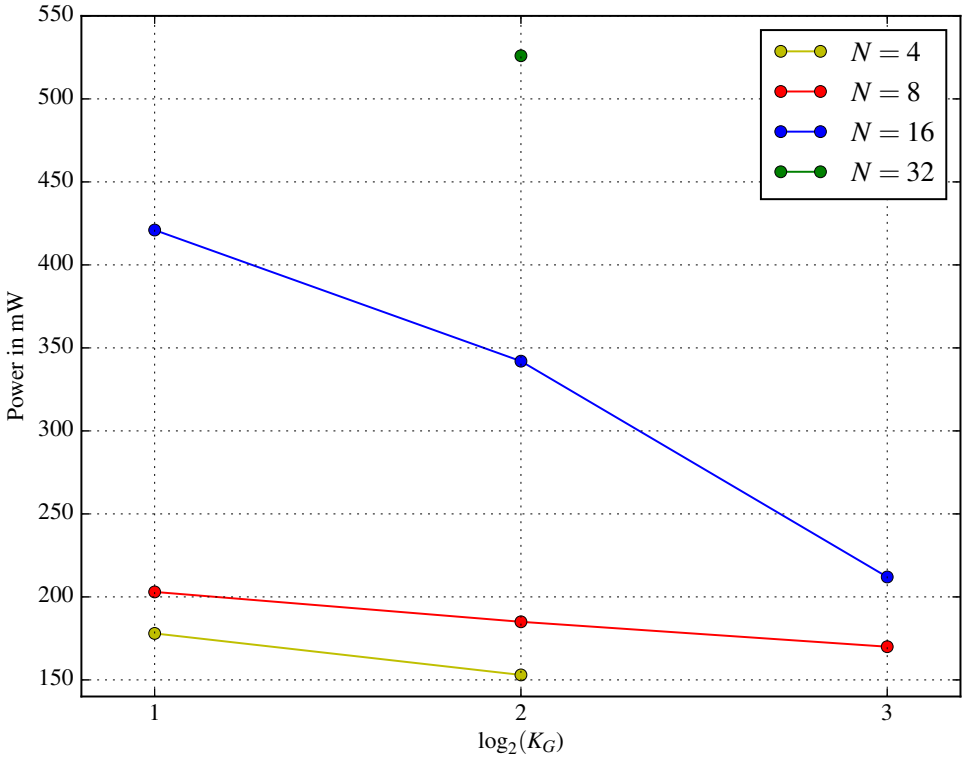


Figure 5.3: Power Consumption estimates for several Network Sizes N and resource sharing level K_G

	$K_G = 2$	$K_G = 4$	$K_G = 8$
$N = 4$	6.87%	6.04%	N.A.
$N = 8$	14.64%	13.03%	14.11%
$N = 16$	28.97%	27.72%	29.85%
$N = 32$	N.A.	91.09%	N.D.

Table 5.1: LUT usage for different N and K_G in the XC7Z020

with increasing K_G, it is clear, and expectable, that the LUT usage increases with the size of the network by an approximate factor of 2, from $N = 2$ to $N = 16$. As for $N = 32$, the usage does not follow this apparent trend, and rises sharply to 91%. For $K_G = 8$ the usage surpasses the maximum amount of LUTs available in the XC7Z020.

As for LUTs, the FF usage also scales accordingly to a $2\times$ factor. In terms of LUTRAM, used to store the network weights, the amount used increases by 2 with increasing values of N, as before, and **does not** depend on K_G. This is because the amount of weights only depends on the network sizes M and N, and not on K_G. Furthermore, unlike LUTs, it scales well with increasing network sizes, and does not pose a limitation on the network size. The usage results for the XC7Z020 are reported in Table 5.2, and in Table 5.3 the usages for the VC707.

5.2 Validation and Comparison

Over the course of this Section, the functionality of the network is verified against the Python reference module of 5.2.1 that was developed as a reference, both for the forward propagation of the network, as well as for the training algorithm. The methodology of this procedure is outlined in Section 5.2.2. On Section 5.2.3, the performance results of Section 5.1.1 is translated into how many classifications per second this design achieves, and how that figure compares with the capabilities of the Python module and the current existing hardware implementation [5].

5.2.1 Reference Python Module

Before performing the Verilog description of the network, the ideas outlined in Section 2 were tested first by building a software version of an LSTM Network. Since we want to quickly test the ideas without much effort, Python and Numpy were used rather than MATLAB, since Python/Numpy has higher performance at the same level of code complexity. The code for this

	LUTRAM	FF
$N = 4$	0.18%	3.39%
$N = 8$	4.41%	6.7%
$N = 16$	8.83%	13.36%
$N = 32$	17.66%	26.45%

Table 5.2: Flip-flop and LUTRAM usage for the XC7Z020

	LUTRAM	FF	LUT
$N = 64$	24.22%	9.14%	24.22%
$N = 128$	14.09%	18.3%	41.82%

Table 5.3: Flip-flop, LUTRAM and LUT usage for the VC707

Python class is listed in Appendix B. It supports both the forward propagation of an already trained network, but also provides functions to train a model using the SPSA method outlined in Section 2.1.4.2 and also Backpropagation Through-Time as presented in [2]. As the reader can attest by running the code himself, the SPSA proves to be a valid approach for training these networks, achieving convergence for the Machine Learning task that will be presented in the next paragraph, although it converges slower than Backpropagation Through Time.

The learning problem presented to both the software and hardware network is the sum of two binary numbers of 8 bits. The bits of each number at position i are input to the network as a vector, and the network outputs its prediction of the correct value of the i-th bit of the *result* number. After the whole number number is processed, the memory cells of the LSTM network are reset and a new addition task can be presented to the network. Even though this seems a rather simple problem, it accounts for all the essential issues at which this network excels: first, this is a *classification* problem in which the Machine Learning algorithm needs to output a prediction based on the input feature vector, and that prediction has to take into account the *history* of predictions so far, because the current bit is the sum of not only the bits of the two operands, but also the *carry* generated at the last few positions – this is where the memory cells of this special Neural Network come into effect.

5.2.2 Methodology

The Python testbench that both trains and verifies the predictions of the software network of Appendix B is listed in Appendix C. The scripts train the network for a maximum of 100 epochs, and in each epoch, the network is presented with 5000 8 bit additions for training, and then 100 8 bit additions for testing, where the prediction error is calculated, and the training convergence is verified (when there is zero error for two consecutive times). After achieving convergence, the script stores the trained network weights in Pickle format. These weights are, then, converted to $Q6.11$ fixed-point format and directly loaded into the Verilog model by the Verilog testbench. The Verilog testbench then applies in order the bits of randomly generated numbers, captures the output bit and checks whether it is correct or not, and counts the number of incorrect bits.

This network has, then, two inputs ($M = 2$) and eight neurons ($N = 8$). Furthermore, the output of the eight LSTM neurons is reduced to a single output by means of a simple neuron, as the one described in Figure 2.1a. This introduces a further 5 clock cycle overhead that is not accounted in Section 5.2.3, since it is not actually part of the LSTM Network. Although it was placed in series in the datapath, after the LSTM network, it can be run in parallel with the LSTM network, introducing no overhead whatsoever.

	$K_G = 8$	$K_G = 4$	$K_G = 2$	Python	Speed-up
$N = 4$	N.A.	309.68 ns	259.12 ns	65 μs	$\times$251
$N = 8$	793.46 ns	421.12 ns	317.52 ns	72 μs	$\times$228
$N = 16$	1.497 μs	738.19 ns	461.336 ns	96 μs	$\times$208
$N = 32$	N.D.	1.586 μs	N.A.	185 μs	$\times$117

Table 5.4: Total processing time for a single forward propagation on the XC7Z020

A Python script was written (see Appendix D) to automatically compile the Verilog code, generate the input and output data, and call Questasim to run the simulation of the HDL code. Figure 5.4 shows the waveform viewer window of a run of the Verilog testbench, and in Appendix E shows the transcript of a successful run of this testbench, proving that the learning ability of the Hardware network still retains the predicting capability of the software network.

5.2.3 Performance

To evaluate the performance, a metric was defined based on how many predictions it can produce per second (i.e. produce a new result bit in the output sequence), in **millions**. The metric on [5] is not very informative, since although a system can perform many calculations per second, if those operations are redundant, that metric has no relevant information regarding **how fast** the system can **perform the actual task it was meant to do**, which in this case is a complete Forward Propagation through the network. The prediction time is the time elapsed since a new input vector is applied *to* the moment the LSTM Network produces an output vector. Hence, we multiply the number of clock cycles yielded by Equation 4.13 by the equivalent clock period from the synthesis clock report of Figure 5.1. This result is epitomized in Table 5.4, where the calculation time of the Python module's Forward Propagation function is also included. This time was measured using the **timeit**[1] module, that allows the evaluation of the execution time of small pieces of code as well as complete functions with arguments. The Python code was run on a Linux System, powered by an i7-3770k Intel Processor, running at 4.2GHz.

The performance increase is significant, even for the slowest of the designs (the $N = 32$ network). The hardware network is, at best, $\times$251 faster than the software counterpart, and at worst $\times$117 faster. Also, it is noticeable that increasing the level of resource sharing increases the computation time, since the level of parallelism is lower.

Since the previous values are the time needed for a *single* forward propagation, to know how many forward propagations we can perform *per second*, we only need to invert the previous values. These values are presented in Figure 5.5. While the $N = 8$ and $K_G = 2$ network is able to perform around 3.15 million predictions per second, the Python model can only output around 14 thousand predictions, which is a very significant result that proves the relevance of this implementation.

As for the larger-sized networks synthesized in the VC707, the results are also very promising. For network sizes of $N = 64$ and $N = 128$, a complete forward propagation takes 1.14 μs and 2.052

[1] https://docs.python.org/3.5/library/timeit.html

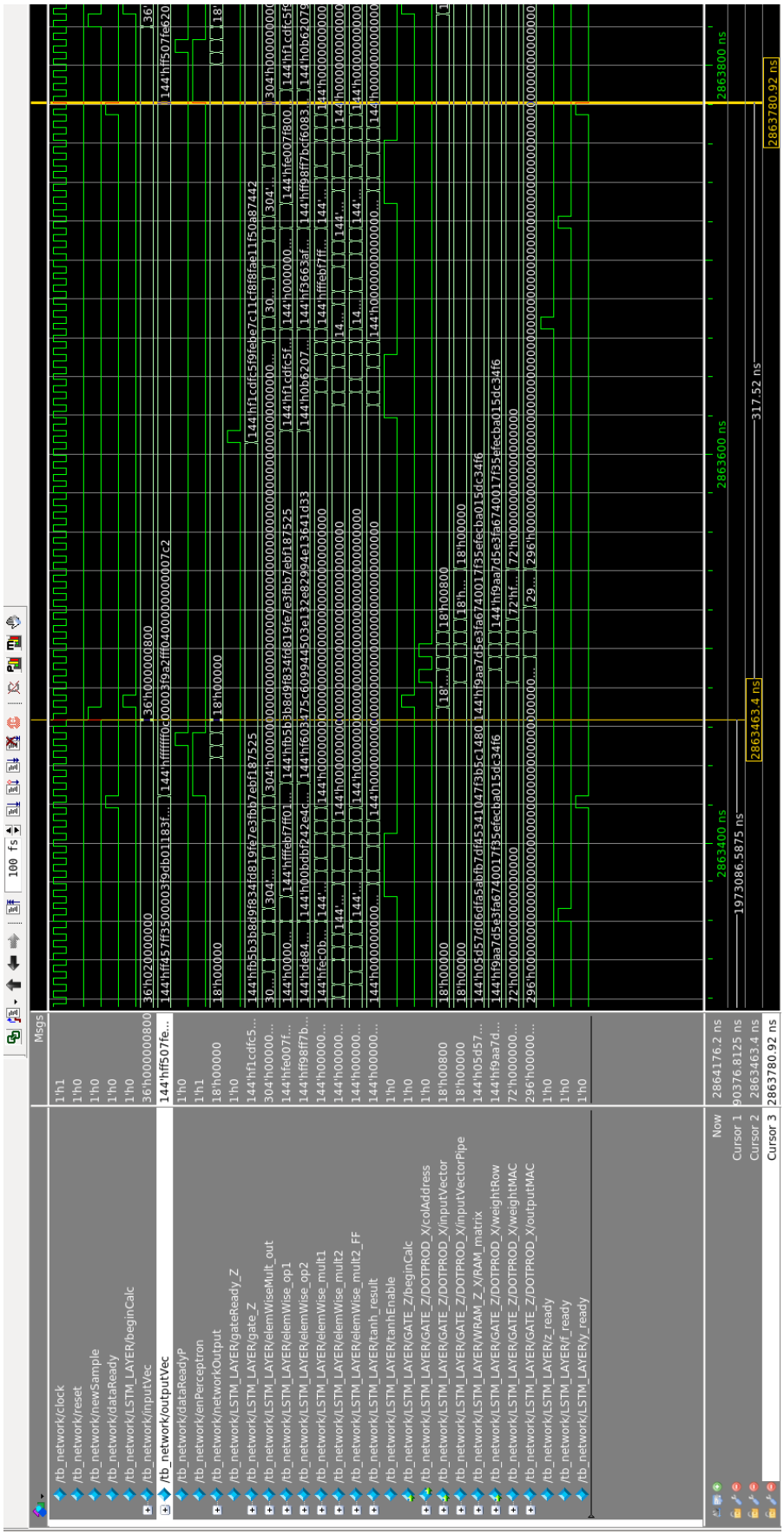


Figure 5.4: Screenshot of the simulation run of a complete forward propagation for $N = 8$ and $K_G = 2$. The period between the two time cursors is the same as the one reported in Table 5.4

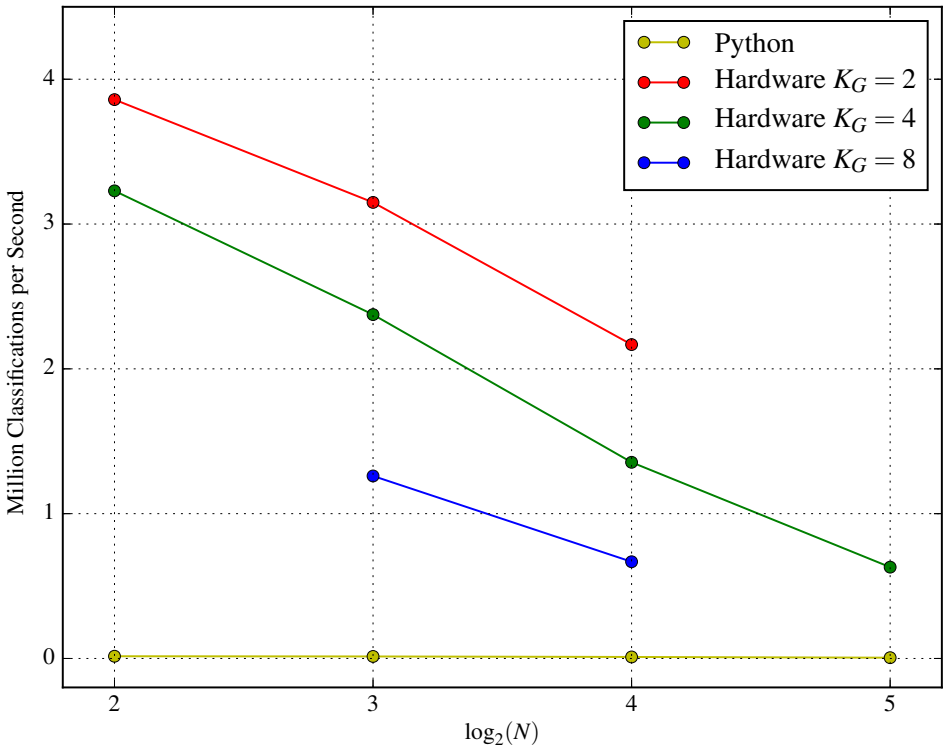


Figure 5.5: Millions of classifications per second of each design according to the network size N. The comparison is between the software Python model and 3 networks of different levels of resource sharing K_G.

μs respectively, and for both the maximum clock frequency achievable was 140.845 MHz. Since the design [5], for $N = 128$, takes an estimated 29.13 μs, this design yields an improvement of 14$\times$ over it.

Chapter 6

Conclusion

The LSTM Hardware architecture presented surpassed the performance of the custom-built software implementation by 251×, at best, and also the only current hardware implementation by 14×, and solely making use of internal FPGA resources, achieving a higher level of parallelism. The higher levels of parallelism of this work are achieved at the cost of increasing design complexity, which undermines its scalability for higher sized networks, unlike the implementation of Chang et al. [5]. On the other hand, the HDL description of this work is parameterized, and is thus very flexible for networks of any size, not requiring a redesign of the system every time a different sized network is required. Furthermore, making use of internal memory makes it suitable for including an on-chip learning system that can perform training on the network weights.

Given these results, this architecture places itself as the current state of the art system in LSTM Neural Networks implementations in hardware, providing the most efficient implementation to date.

A possible follow-up of the work of this thesis, is the integration of the training circuit in Section 4.3.2, providing an LSTM Network with on-chip training capabilities.

Appendix A

Remez Algorithm Implementation

```python
"""
Remez Algorithm for minimax polynomial approximation of transcendental functions

Jose Pedro Castro Fonseca, 2016

University of Porto, Portugal

"""

import numpy as np
import matplotlib.pyplot as plt
from scipy.signal import argrelmax

# The target approximation function
def f(x):
    return np.tanh(x)

# Nber of iterations
numIter = 8

# Approximation Interval
b = 3
a = 1

# Fixed-point system precision
prec = 2**(-10)
t = np.arange(a, b, prec)

# Polynomial Degree
n = 2

# Initial set of points
x = np.zeros((n+2,1))
for i in reversed(range(n+2)): x[n+1-i,0] = (a+b)/2+(b-a)/2*np.cos(i*np.pi/(n+1))

# The numerical matrices
A = np.zeros((n+2, n+2))
A[:,0] = 1
for i in range(n+2): A[i,n+1] = (-1)**(i+1)

```

```
c = np.zeros((n+2,1))

# The algorithm iterations
for it in range(numIter):
    # Compute the A matrix
    for i in range(1,n+1):
        A[:,i] = (x**i).T
    # Compute the b matrix
    for i in range(n+2): c[i] = f(x[i])

    # Solve the system
    p = np.dot( np.linalg.inv(A), c )

    # Recompute the x point vector
    pol = np.poly1d(np.reshape(np.flipud(p[0:n+1]).T, (n+1)))
    diff = abs(pol(t) - f(t))

    extremaIndices = argrelmax(abs(diff), axis=0, mode='wrap')[0]
    if len(extremaIndices) < (n+2) :
        extremaIndices = np.resize(extremaIndices, (1,n+2))
        if abs(diff[t[len(t)-1]]) > abs(diff[t[len(t)-2]]):
            extremaIndices[0,n+1] = len(t)-1
        else:
            extremaIndices = np.roll(extremaIndices, 1)
            extremaIndices[0,0] = 0

    x = t[extremaIndices].T
    #Prints Progress
    print("Error: ", max(abs(diff)))


print("************************")
print("Coefficients: ", p[0:n+1].T)
print("Error: ", max(abs(diff)))
#print(extremaIndices)
plt.figure(1)
plt.plot(t, pol(t), t, f(t))

plt.figure(2)
plt.plot(t, diff)
plt.show()
```

Listing A.1: Python script that implements Remez's algorithm

Appendix B

LSTM Layer Implementation

```
# -*- coding: utf-8 -*-
"""
LSTM Layer Class
@author: Jose Pedro Castro Fonseca, University of Porto, PORTUGAL
"""
import numpy as np
import time

# Defines the seed for random() as the current time in seconds
np.random.seed(round(time.time()))

# Evaluates the Sigmoid Function
def sigmoid(x):
    output = 1/(1+np.exp(-x))
    return output

# Evaluates the derivative of the Sigmoid Func, based on a previous sigmoid() call
def sigmoidPrime(output):
    return output*(1-output)

# Evaluates the derivative of the TanH Func, based on a previous np.tanh() call
def tanhPrime(output):
    return (1-output**2)

class LSTMlayer :

    def __init__(self, inputUnits, hiddenUnits, learnRate, T):
        # The Network Parameters, passed by the user
        self.inputUnits  = inputUnits
        self.hiddenUnits = hiddenUnits
        self.learnRate   = learnRate
        self.T           = T
        self.t           = 0

        # Initializing the matrix weights
        #LSTM Block
        self.Wz = np.random.random((hiddenUnits, inputUnits)) - 0.5
        self.Wi = np.random.random((hiddenUnits, inputUnits)) - 0.5
        self.Wf = np.random.random((hiddenUnits, inputUnits)) - 0.5
        self.Wo = np.random.random((hiddenUnits, inputUnits)) - 0.5
```

```
        self.Rz = np.random.random((hiddenUnits, hiddenUnits)) - 0.5
        self.Ri = np.random.random((hiddenUnits, hiddenUnits)) - 0.5
        self.Rf = np.random.random((hiddenUnits, hiddenUnits)) - 0.5
        self.Ro = np.random.random((hiddenUnits, hiddenUnits)) - 0.5

        self.pi = np.random.random((hiddenUnits)) - 0.5
        self.pf = np.random.random((hiddenUnits)) - 0.5
        self.po = np.random.random((hiddenUnits)) - 0.5

        self.bz = np.random.random((hiddenUnits)) - 0.5
        self.bi = np.random.random((hiddenUnits)) - 0.5
        self.bo = np.random.random((hiddenUnits)) - 0.5
        self.bf = np.random.random((hiddenUnits)) - 0.5

        # Updates
        self.Wz_update = np.zeros_like(self.Wz)
        self.Wi_update = np.zeros_like(self.Wi)
        self.Wf_update = np.zeros_like(self.Wf)
        self.Wo_update = np.zeros_like(self.Wo)

        self.Rz_update = np.zeros_like(self.Rz)
        self.Ri_update = np.zeros_like(self.Ri)
        self.Rf_update = np.zeros_like(self.Rf)
        self.Ro_update = np.zeros_like(self.Ro)

        self.po_update = np.zeros_like(self.po)
        self.pf_update = np.zeros_like(self.pf)
        self.pi_update = np.zeros_like(self.pi)

        self.bz_update = np.zeros_like(self.bz)
        self.bi_update = np.zeros_like(self.bi)
        self.bf_update = np.zeros_like(self.bf)
        self.bo_update = np.zeros_like(self.bo)

        # State vars
        self.y_prev = np.zeros((hiddenUnits,self.T))
        self.x_prev = np.zeros((inputUnits,self.T))
        self.o_prev = np.zeros((hiddenUnits,self.T))
        self.f_prev = np.zeros((hiddenUnits,self.T))
        self.c_prev = np.zeros((hiddenUnits,self.T))
        self.z_prev = np.zeros((hiddenUnits,self.T))
        self.i_prev = np.zeros((hiddenUnits,self.T))

        self.delta_y_list = np.zeros((hiddenUnits,self.T))
        self.delta_o_list = np.zeros((hiddenUnits,self.T))
        self.delta_c_list = np.zeros((hiddenUnits,self.T))
        self.delta_f_list = np.zeros((hiddenUnits,self.T))
        self.delta_i_list = np.zeros((hiddenUnits,self.T))
        self.delta_z_list = np.zeros((hiddenUnits,self.T))

        self.future_y = np.zeros_like(self.y_prev[:,1])
        self.future_c = np.zeros_like(self.c_prev[:,1])

        # Delta vectors for a previous layer
        self.delta_x = np.zeros((inputUnits,self.T))

```

```
    def forwardPropagate(self, X):

        # Note that in a list, accessing [-1] corresponds to the last element appended. so y_prev[-1] == y^(t-1)
        if (self.t != 0):
            self.z_prev[:,self.t] = np.tanh( np.dot(self.Wz,X) + np.dot(self.Rz,self.y_prev[:,self.t-1]) + self.bz.T )
            self.i_prev[:,self.t] = sigmoid( np.dot(self.Wi,X) + np.dot(self.Ri,self.y_prev[:,self.t-1]) + np.multiply(self.pi,self.c_prev[:,self.t-1]) + self.bi.T )
            self.f_prev[:,self.t] = sigmoid( np.dot(self.Wf,X) + np.dot(self.Rf,self.y_prev[:,self.t-1]) + np.multiply(self.pf,self.c_prev[:,self.t-1]) + self.bf.T )
            self.c_prev[:,self.t] = np.multiply(self.z_prev[:,self.t],self.i_prev[:,self.t]) + np.multiply(self.c_prev[:,self.t-1],self.f_prev[:,self.t])
            self.o_prev[:,self.t] = sigmoid( np.dot(self.Wo,X) + np.dot(self.Ro,self.y_prev[:,self.t-1]) + np.multiply(self.po,self.c_prev[:,self.t]) + self.bo.T )
            self.y_prev[:,self.t] = np.multiply(np.tanh(self.c_prev[:,self.t]), self.o_prev[:,self.t])
        else:
            self.z_prev[:,self.t] = np.tanh( np.dot(self.Wz,X) + np.dot(self.Rz,self.future_y) + self.bz.T )
            self.i_prev[:,self.t] = sigmoid( np.dot(self.Wi,X) + np.dot(self.Ri,self.future_y) + self.bi.T )
            self.f_prev[:,self.t] = sigmoid( np.dot(self.Wf,X) + np.dot(self.Rf,self.future_y) + self.bf.T )
            self.c_prev[:,self.t] = np.multiply(self.z_prev[:,self.t],self.i_prev[:,self.t]) + np.multiply(self.future_c,self.f_prev[:,self.t])
            self.o_prev[:,self.t] = sigmoid( np.dot(self.Wo,X) + np.dot(self.Ro,self.future_y) + np.multiply(self.po, self.c_prev[:,self.t]) + self.bo.T )
            self.y_prev[:,self.t] = np.multiply(np.tanh(self.c_prev[:,self.t]), self.o_prev[:,self.t])

        self.x_prev[:,self.t] = X

        if (self.t == self.T-1) :
            self.future_y = self.y_prev[:,self.t]  # Saving the state of these variables between iterations, otherwise y_prev will be zero!
            self.future_c = self.c_prev[:,self.t]
            self.t = 0                             # Reset the time pointer within time frame
            return self.y_prev[:,self.T-1]
        else:
            self.t += 1
            return self.y_prev[:,self.t-1]



    def backPropagate_T(self, upperLayerDeltas):
        # EVALUATING THE DELTAS
        for t in reversed(range(self.T)):
            if (t == self.T-1):
                self.delta_y_list[:,t] = (upperLayerDeltas[:,t])
                self.delta_o_list[:,t] = (np.multiply( np.multiply(upperLayerDeltas[:,t],np.tanh(self.c_prev[:,t])), sigmoidPrime(self.o_prev[:,t])))
                self.delta_c_list[:,t] = (np.multiply(np.multiply(self.delta_y_list[:,t], tanhPrime(np.tanh(self.c_prev[:,t]))), self.o_prev[:,t]))
            else:
```

```
137                 self.delta_y_list[:,t] = (upperLayerDeltas[:,t] + np.dot(self.Rz,self.
    delta_z_list[:,t+1]) + np.dot(self.Ri,self.delta_i_list[:,t+1]) + np.dot(self.Rf,self.
    delta_f_list[:,t+1]) + np.dot(self.Ro,self.delta_o_list[:,t+1]))
138                 self.delta_o_list[:,t] = (np.multiply( np.multiply(upperLayerDeltas[:,t],np.
    tanh(self.c_prev[:,t])), sigmoidPrime(self.o_prev[:,t])))
139                 self.delta_c_list[:,t] = (np.multiply(np.multiply(self.delta_y_list[:,t],
    tanhPrime(np.tanh(self.c_prev[:,t]))), self.o_prev[:,t]) + np.multiply(self.delta_c_list
    [:,t+1],self.f_prev[:,t+1]))
140
141             if (t != 0) :
142                 self.delta_f_list[:,t] = (np.multiply(np.multiply(self.delta_c_list[:,t],self
    .c_prev[:,t-1]), sigmoidPrime(self.f_prev[:,t])))
143
144             self.delta_i_list[:,t] = (np.multiply( np.multiply(self.delta_c_list[:,t],self.
    z_prev[:,t]), sigmoidPrime(self.i_prev[:,t])))
145             self.delta_z_list[:,t] = (np.multiply( np.multiply(self.delta_c_list[:,t],self.
    i_prev[:,t]), tanhPrime(self.z_prev[:,t])))
146             self.delta_x[:,t] = np.dot(self.Wz.T,self.delta_z_list[:,t]) + np.dot(self.Wi.T,
    self.delta_i_list[:,t]) + np.dot(self.Wf.T,self.delta_f_list[:,t]) + np.dot(self.Wo.T,
    self.delta_o_list[:,t])
147
148         # EVALUATING THE UPDATES
149         for t in range(self.T):
150             self.Wz += self.learnRate * np.outer(self.delta_z_list[:,t],self.x_prev[:,t])
151             self.Wi += self.learnRate * np.outer(self.delta_i_list[:,t],self.x_prev[:,t])
152             self.Wf += self.learnRate * np.outer(self.delta_f_list[:,t],self.x_prev[:,t])
153             self.Wo += self.learnRate * np.outer(self.delta_o_list[:,t],self.x_prev[:,t])
154             self.po += self.learnRate * np.multiply(self.c_prev[:,t],self.delta_o_list[:,t])
155             self.bz += self.learnRate * self.delta_z_list[:,t]
156             self.bi += self.learnRate * self.delta_i_list[:,t]
157             self.bf += self.learnRate * self.delta_f_list[:,t]
158             self.bo += self.learnRate * self.delta_o_list[:,t]
159
160             if(t < self.T-1):
161                 self.Rz += self.learnRate * np.outer(self.delta_z_list[:,t+1],self.y_prev[:,t
    ])
162                 self.Ri += self.learnRate * np.outer(self.delta_i_list[:,t+1],self.y_prev[:,t
    ])
163                 self.Rf += self.learnRate * np.outer(self.delta_f_list[:,t+1],self.y_prev[:,t
    ])
164                 self.Ro += self.learnRate * np.outer(self.delta_o_list[:,t+1],self.y_prev[:,t
    ])
165                 self.pi += self.learnRate * np.multiply(self.c_prev[:,t],self.delta_i_list[:,
    t+1])
166                 self.pf += self.learnRate * np.multiply(self.c_prev[:,t],self.delta_f_list[:,
    t+1])
167
168         return  self.delta_x
```

Listing B.1: Python class that implements an LSTM Layer

Appendix C

LSTM Layer Testbench

```
import numpy as np
import matplotlib.pyplot as plt
import time
import sys
import pickle
import LSTMlayer

np.random.seed(round(time.time()))

# compute sigmoid nonlinearity
def sigmoid(x):
    output = 1/(1+np.exp(-x))
    return output

# convert output of sigmoid function to its derivative
def sigmoid_output_to_derivative(output):
    return output*(1-output)

def sigmoidPrime(output):
    return output*(1-output)

def tanhPrime(output):
    return (1-output**2)

# training dataset generation
binary_dim = 8
largest_number = pow(2,binary_dim)

# Simulation Parameters
pert  = float(sys.argv[1])
alpha = float(sys.argv[2])
wmax  = float(sys.argv[3])
samplesPerEpoch = 500
input_dim  = 2
hidden_dim = int(sys.argv[4])
output_dim = 1
maxEpoch   = 100
trainSamp  = 5000
testSamp   = 100

```

```
# initialize neural network weights
lstmLayer1 = LSTMlayer.LSTMlayer(input_dim, hidden_dim, output_dim, alpha, 'SPSA', pert, wmax
    , binary_dim)
plt.axis([0, maxEpoch, 0, binary_dim])
plt.title("Alpha={0} -- Perturbation={1} -- Wmax={2}".format(alpha, pert, wmax))
plt.ion()
plt.show()

epochError = 0
correct    = 0
# training logic
for i in range(maxEpoch):

    print("Training Epoch: ", i)

    for j in range(trainSamp):
        # generate a simple addition problem (a + b = c)
        a_int = np.random.randint(largest_number/2) # int version
        a = np.binary_repr(a_int, width=binary_dim) # binary encoding

        b_int = np.random.randint(largest_number/2) # int version
        b = np.binary_repr(b_int, width=binary_dim) # binary encoding

        # true answer
        c_int = a_int + b_int
        c = np.binary_repr(c_int, width=binary_dim)

        # -------------- THE FORWARD PROPAGATION STEP -------------- #
        for position in range(binary_dim):

            # generate input and output
            X = np.array([[int(a[binary_dim - position - 1]), int(b[binary_dim - position -
    1])]]).T
            y = np.array([int(c[binary_dim - position - 1])]).T

            # Perform a forward propagation through the network
            y_pred = lstmLayer1.trainNetwork_SPSA(X, y)

        lstmLayer1.resetNetwork()

    for j in range(testSamp):
        # generate a simple addition problem (a + b = c)
        a_int = np.random.randint(largest_number/2) # int version
        a = np.binary_repr(a_int, width=binary_dim) # binary encoding

        b_int = np.random.randint(largest_number/2) # int version
        b = np.binary_repr(b_int, width=binary_dim) # binary encoding

        # true answer
        c_int = a_int + b_int
        c = np.binary_repr(c_int, width=binary_dim)

        # -------------- THE FORWARD PROPAGATION STEP -------------- #
        for position in range(binary_dim):

            # generate input and output
```

```
            X = np.array([[int(a[binary_dim - position - 1]), int(b[binary_dim - position - 1])]]).T
            y = np.array([int(c[binary_dim - position - 1])]).T

            # Perform a forward propagation through the network
            y_pred = lstmLayer1.forwardPropagate(X)
            lstmLayer1.prev_c = lstmLayer1.c
            lstmLayer1.prev_y = lstmLayer1.y

            # decode estimate so we can print it out
            epochError += int(np.abs(y - np.round(y_pred)))

        lstmLayer1.resetNetwork()

    if (epochError/testSamp  == 0):
        correct += 1
        if(correct == 2):
            print("Convergence Acheived in {0} epochs".format(i-2))
            break
    else:
        correct = 0


    print("Average Error:", epochError/testSamp)
    #plt.scatter(i, epochError/testSamp, linestyle='-.')
    #plt.draw()
    epochError = 0

with open('objs.pickle', 'wb') as f:
    pickle.dump([lstmLayer1.Wz, lstmLayer1.Wi, lstmLayer1.Wf, lstmLayer1.Wo, lstmLayer1.Rz, lstmLayer1.Ri, lstmLayer1.Rf, lstmLayer1.Ro, lstmLayer1.bz, lstmLayer1.bi, lstmLayer1.bf, lstmLayer1.bo,], f)

with open('layer.pickle', 'wb') as f:
    pickle.dump(lstmLayer1, f)

print("Epochs: {0}".format(i+1))
wait = input("PRESS ENTER TO CONTINUE.")
```

Listing C.1: Python script that tests the LSTM Layer of Listing B.1

Appendix D

Verification script for the Verilog simulation of the network

```
import pickle
import numpy as np
import os

def real_to_Qnm(real, n, m):
    if(real >= 0):
        return int(np.round(real*(2**m)).astype(int)) & int(2**(n+m+1)-1)
    else:
        return int(2**(n+m+1) + np.round(real*(2**m)).astype(int)) & int(2**(n+m+1)-1)
def Qnm_to_real(real, n, m):
    real = int(real) & int(2**(n+m+1)-1)
    if(real >= 2**(n+m)):
        return (real-2**(n+m+1))/(2**m)
    else:
        return real/(2**m)
def sigmoid(x):
  output = 1/(1+np.exp(-x))
  return output

numBits  = 8
binary_dim = numBits
largest_number = pow(2,binary_dim)
hiddenSz = 8
QN = 6
QM = 11
numTrain = 1000
prevX = list()
outputGolden  = list()
outputVerilog = list()
outputPython  = list()
wrongBits = 0
f_in  = open("goldenIn_x.bin", "w")




for i in range(numTrain):
```

```
  # Generates the golden input and output
  a_int = np.random.randint(largest_number/2) # int version
  a = np.binary_repr(a_int, width=binary_dim) # binary encoding

  b_int = np.random.randint(largest_number/2) # int version
  b = np.binary_repr(b_int, width=binary_dim) # binary encoding

  c_int = a_int + b_int
  c = np.binary_repr(c_int, width=binary_dim)

  for position in range(binary_dim):
    X  = np.array([[int(a[binary_dim - position - 1]), int(b[binary_dim - position - 1])]]).T
    y  = np.array([int(c[binary_dim - position - 1])]).T
    prevX.append(X)
    f_in.write("{0:018b}\n".format(real_to_Qnm(X[0,0], QN, QM)))
    f_in.write("{0:018b}\n".format(real_to_Qnm(X[1,0], QN, QM)))

f_in.close()

# Compiles and runs the Verilog simulation
os.system("vlog *.v")
os.system("vsim -voptargs=+acc -c -do \"run -all\" tb_network")

# Loads the pickled layer
f_pkl = open("layer.pickle", "rb")
layer = pickle.load(f_pkl);

# Loads the output values
layerOut = np.zeros((8,1));
fout = open("output.bin", "r");

layer.resetNetwork()

for n in range(numTrain):
  for i in range(numBits):
    line = fout.readline();
    outputNetwork = Qnm_to_real(int(line), QN,QM);

    # HDL Network output
    temp = layer.forwardPropagate(prevX[i+numBits*n])
    outputVerilog.append(int(np.round(sigmoid(outputNetwork))));
    outputPython.append(int(np.round(temp)));

    if(outputVerilog[i] ^ outputPython[i]) :
      print("Error: ", outputVerilog, " --> ", outputPython)
      print(sigmoid(outputNetwork), " --> ", temp)
      wrongBits += 1

    layer.prev_c = layer.c
    layer.prev_y = layer.y

  outputVerilog.reverse()
  outputPython.reverse()

  #print("Sample %d", n)
  #print(outputVerilog, "\n", outputPython);
  outputVerilog = list()
```

```
95    outputPython = list()
96
97    layer.resetNetwork()
98
99  print("% wrong bits: ", 100*wrongBits/(numTrain*numBits))
```

Listing D.1: Python script that automates the verification procedure

Appendix E

Transcript from a successful simulation of the Hardware Network

```
QuestaSim-64 vlog 10.4c_5 Compiler 2015.11 Nov 14 2015
Start time: 20:23:41 on Jun 26,2016
vlog array_prod.v dot_prod.v gate.v network.v sigmoid.v tanh.v tb_network.v tb_top_network.v
    test.v top_network.v weightRAM.v
-- Compiling module array_prod
-- Compiling module dot_prod
-- Compiling module gate
-- Compiling module network
-- Compiling module sigmoid
-- Compiling module tanh
-- Compiling module tb_network
-- Compiling module tb_top_network
-- Compiling module test
-- Compiling module top_network
-- Compiling module weightRAM

Top level modules:
  tb_network
  tb_top_network
  test
End time: 20:23:42 on Jun 26,2016, Elapsed time: 0:00:01
Errors: 0, Warnings: 0
Reading pref.tcl

# 10.4c_5

# vsim -voptargs="+acc" -c -do "run -all" tb_network
# Start time: 20:23:44 on Jun 26,2016
# ** Note: (vsim-8009) Loading existing optimized design _opt7
# //  Questa Sim-64
# //  Version 10.4c_5 linux_x86_64 Nov 14 2015
# //
# //  Copyright 1991-2015 Mentor Graphics Corporation
# //  All Rights Reserved.
# //
# //  THIS WORK CONTAINS TRADE SECRET AND PROPRIETARY INFORMATION
# //  WHICH IS THE PROPERTY OF MENTOR GRAPHICS CORPORATION OR ITS
```

```
# //  LICENSORS AND IS SUBJECT TO LICENSE TERMS.
# //  THIS DOCUMENT CONTAINS TRADE SECRETS AND COMMERCIAL OR FINANCIAL
# //  INFORMATION THAT ARE PRIVILEGED, CONFIDENTIAL, AND EXEMPT FROM
# //  DISCLOSURE UNDER THE FREEDOM OF INFORMATION ACT, 5 U.S.C. SECTION 552.
# //  FURTHERMORE, THIS INFORMATION IS PROHIBITED FROM DISCLOSURE UNDER
# //  THE TRADE SECRETS ACT, 18 U.S.C. SECTION 1905.
# //
# Loading work.tb_network(fast)
# Loading work.network(fast)
# Loading work.gate(fast)
# Loading work.dot_prod(fast)
# Loading work.dot_prod(fast__1)
# Loading work.weightRAM(fast)
# Loading work.weightRAM(fast__1)
# Loading work.sigmoid(fast)
# Loading work.tanh(fast)
# Loading work.array_prod(fast)
# run -all
# Simulation started at 0.000000
# Input Sample           0
# Input Sample         100
# Input Sample         200
# Input Sample         300
# Input Sample         400
# Input Sample         500
# Input Sample         600
# Input Sample         700
# Input Sample         800
# Input Sample         900
# ** Note: $stop    : tb_network.v(216)
#    Time: 1636005 ns  Iteration: 0  Instance: /tb_network
# Break in Module tb_network at tb_network.v line 216
# Stopped at tb_network.v line 216
VSIM 2>
# End time: 20:23:52 on Jun 26,2016, Elapsed time: 0:00:08
# Errors: 0, Warnings: 0
Error:  [0, 1, 1, 1, 0, 0]  -->  [0, 1, 1, 1, 0, 1]
0.46793955524  -->  [[ 0.50527151]]
Error:  [0, 1, 1, 1, 0, 0]  -->  [0, 1, 1, 1, 0, 1]
0.46793955524  -->  [[ 0.50527151]]
% wrong bits:  0.025
```

Listing E.1: Command line transcript of the python script that runs and validates the Hardware network

References

[1] Alex Graves and Jürgen Schmidhuber. Framewise phoneme classification with bidirectional lstm and other neural network architectures. *Neural Networks*, pages 5–6, 2005.

[2] Klaus Greff, Rupesh Kumar Srivastava, Jan Koutník, Bas R. Steunebrink, and Jürgen Schmidhuber. LSTM: A search space odyssey. *CoRR*, abs/1503.04069, 2015.

[3] Thomas E. Tkacik. *A Hardware Random Number Generator*, pages 450–453. Springer Berlin Heidelberg, Berlin, Heidelberg, 2003.

[4] S. Hochreiter and J. Schmidhuber. Long short-term memory. *Neural Computation*, 9(8):1735 – 80, 1997.

[5] Andre Xian Ming Chang, Berin Martini, and Eugenio Culurciello. Recurrent Neural Networks Hardware Implementation on FPGA. *CoRR*, abs/1511.05552, 2015.

[6] Christopher M. Bishop. *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006.

[7] Yoshua Bengio. *Artificial Neural Networks and Their Application to Sequence Recognition*. PhD thesis, McGill University, Montreal, Que., Canada, Canada, 1991. UMI Order No. GAXNN-72116 (Canadian dissertation).

[8] Sepp Hochreiter, Yoshua Bengio, Paolo Frasconi, and Jürgen Schmidhuber. Gradient Flow in Recurrent Nets: the Difficulty of Learning Long-Term Dependencies. *A Field Guide to Dynamical Recurrent Neural Networks, IEEE Press*, 2001.

[9] Yoshua Bengio, Patrice Simard, and Paolo Frasconi. Learning long-term dependencies with gradient descent is difficult. *IEEE Transactions on Neural Networks*, 5(2):157 – 166, 1994.

[10] F.A. Gers, J. Schmidhuber, and F. Cummins. Learning to forget: continual prediction with LSTM. *Neural Computation*, 12(10):2451 – 71, 2000.

[11] F.A. Gers and J. Schmidhuber. Recurrent nets that time and count. *Proceedings of the IEEE-INNS-ENNS International Joint Conference on Neural Networks. IJCNN 2000. Neural Computing: New Challenges and Perspectives for the New Millennium*, vol.3:189 – 94, 2000.

[12] J.C. Spall. Multivariate stochastic approximation using a simultaneous perturbation gradient approximation. *IEEE Transactions on Automatic Control*, 37(3):332–341, Mar 1992.

[13] Y. Maeda and M. Wakamura. Simultaneous perturbation learning rule for recurrent neural networks and its FPGA implementation. *IEEE Transactions on Neural Networks*, 16(6):1664 – 72, 2005.

[14] A. Graves, M. Liwicki, S. Fernandez, R. Bertolami, H. Bunke, and J. Schmidhuber. A Novel Connectionist System for Unconstrained Handwriting Recognition. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 31(5):855–868, May 2009.

[15] E. Grosicki and H. El Abed. ICDAR 2009 Handwriting Recognition Competition. pages 1398–1402, July 2009.

[16] Thomas M. Breuel, Adnan Ul-Hasan, Mayce Ali Al-Azawi, and Faisal Shafait. High-performance OCR for printed english and fraktur using LSTM networks. In *Proceedings of the 2013 12th International Conference on Document Analysis and Recognition*, ICDAR '13, pages 683–687, Washington, DC, USA, 2013. IEEE Computer Society.

[17] A. Graves, A.-R. Mohamed, and G. Hinton. Speech recognition with deep recurrent neural networks. In *2013 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 6645–6649, May 2013.

[18] Hasim Sak, Andrew W. Senior, and Françoise Beaufays. Long short-term memory recurrent neural network architectures for large scale acoustic modeling. pages 338–342, 2014.

[19] Alex Graves. Generating Sequences with Recurrent Neural Networks. *CoRR*, abs/1308.0850, 2013.

[20] Ilya Sutskever, Oriol Vinyals, and Quoc V. Le. Sequence to sequence learning with neural networks. volume 4, pages 3104 – 3112, Montreal, QC, Canada, 2014.

[21] Sepp Hochreiter, Martin Heusel, and Klaus Obermayer. Fast model-based protein homology detection without alignment. *Oxford Journal in Bioinformatics*, 23(14):1728–1736, JUL 15 2007.

[22] Soren Kaae Sonderby, Casper Kaae Sonderby, Henrik Nielsen, and Ole Winther. Convolutional LSTM networks for subcellular localization of proteins. volume 9199, pages 68 – 80, Mexico City, Mexico, 2015.

[23] B. Lehner, G. Widmer, and S. Bock. A low-latency, real-time-capable singing voice detection method with LSTM recurrent neural networks. pages 21 – 5, Piscataway, NJ, USA, 2015.

[24] Sebastian Bock and Markus Schedl. Polyphonic piano note transcription with recurrent neural networks. *ICASSP, IEEE International Conference on Acoustics, Speech and Signal Processing - Proceedings*, pages 121 – 124, 2012.

[25] Andres E. Coca, Debora C. Correa, and Liang Zhao. Computer-aided music composition with LSTM neural network and chaotic inspiration. Dallas, TX, United states, 2013.

[26] D. Eck and J. Schmidhuber. Finding temporal structure in music: Blues improvisation with LSTM recurrent networks. *Neural Networks for Signal Processing - Proceedings of the IEEE Workshop*, 2002-January:747 – 756, 2002.

[27] Oriol Vinyals, Alexander Toshev, Samy Bengio, and Dumitru Erhan. Show and tell: A neural image caption generator. *CoRR*, abs/1411.4555, 2014.

[28] Subhashini Venugopalan, Huijuan Xu, Jeff Donahue, Marcus Rohrbach, Raymond J. Mooney, and Kate Saenko. Translating videos to natural language using deep recurrent neural networks. *CoRR*, abs/1412.4729, 2014.

[29] Jeff Donahue, Lisa Anne Hendricks, Sergio Guadarrama, Marcus Rohrbach, Subhashini Venugopalan, Kate Saenko, and Trevor Darrell. Long-term Recurrent Convolutional Networks for Visual Recognition and Description. *CoRR*, abs/1411.4389, 2014.

[30] Yutaka Maeda, Hiroaki Hirano, and Yakichi Kanata. A learning rule of neural networks via simultaneous perturbation and its hardware implementation. *Neural Networks*, pages 251–259, 1995.

[31] G. Cauwenberghs. An analog VLSI recurrent neural network learning a continuous-time trajectory. *IEEE Transactions on Neural Networks*, 7(2):346–361, Mar 1996.

[32] Y. Maeda and Rui J.P. de Figueiredo. Learning rules for neuro-controller via simultaneous perturbation. *IEEE Transactions on Neural Networks*, 8(5):1119–1130, 1997.

[33] R. Tavear, J. Dedic, D. Bokal, and A. Zemva. Transforming the LSTM training algorithm for efficient FPGA -based adaptive control of nonlinear dynamic systems. *Informacije MIDEM*, 43(2):131 – 8, 2013.

[34] Randy Yates. Fixed-point arithmetic: An introduction. 2013.

[35] Jean-Michel Muller. *Elementary Functions: Algorithms and Implementation*. Birkhauser, 2nd edition edition, 2005.