FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

Dissertation Preparation

José Pedro Castro Fonseca

Integrated Masters in Electrical and Computer Engineering

Supervisor: João Canas Ferreira

Second Supervisor: Ivo Timóteo

June 20, 2016

Contents

1 Introduction **1**
- 1.1 Background . . . 1
- 1.2 Motivation . . . 2
- 1.3 Objectives . . . 2
- 1.4 People Involved . . . 3

2 Problem Characterization **5**
- 2.1 Theoretical Background . . . 5
  - 2.1.1 Basic Concepts of Machine Learning . . . 5
  - 2.1.2 Artificial Neural Networks . . . 6
  - 2.1.3 Recurrent Neural Networks . . . 10
  - 2.1.4 Long Short-Term Memory Networks . . . 11
- 2.2 Proposed Solution . . . 15

3 State of the Art **17**
- 3.1 LSTM Applications (non-FPGA related) . . . 17
- 3.2 Hardware Implementations of LSTM . . . 18
- 3.3 Training Algorithms and Implementation . . . 19
- 3.4 Final Overview . . . 20

4 Proposed Architecture **21**
- 4.1 Constituent Modules . . . 21
  - 4.1.1 Finite-precision representation system . . . 21
  - 4.1.2 Non-linearity Calculator . . . 23
  - 4.1.3 Matrix-vector Dot Product Unit . . . 27
  - 4.1.4 Dual-port Block RAM . . . 31
  - 4.1.5 Gate Module . . . 31
- 4.2 Hardware LSTM Network – Forward Propagation . . . 32
  - 4.2.1 Network Structure . . . 32
  - 4.2.2 Timing Diagrams of Operation . . . 35
- 4.3 Hardware LSTM Network – SPSA Training . . . 35
  - 4.3.1 Timing Diagrams of Operation . . . 35

5 Work Plan **37**

References **39**

List of Figures

2.1 Representations of a Neuron in an ANN 2.1a and a biological multi polar neuron 2.1b 7
2.2 Possible activation functions of choice for a neuron 8
2.3 Three layer ANN 9
2.4 Representation of a Recurrent Neuron (Figure 2.4a) and a Recurrent Neuron unfolded through time (Figure 2.4b) 11
2.5 A complete LSTM neuron, with all the features as described in [1]. Source: [2] . 12

4.1 Block Diagram of the Non-linearity Calculator 27
4.2 The output of the HDL implementation of the activation functions 28
4.3 A column of the matrix that serves as input to the module 30
4.4 The *i*-th row multiplication unit of the Module 30
4.5 Diagram of the hardware block that implements the Gate 32
4.6 Hardware LSTM Network with full level of parallelism 33
4.7 Optimised version of 4.6 that shares the elementwise multiplication and the activation function blocks 34

Abbreviations and Symbols

ANN	Artificial Neural Networks
BPTT	Backpropagation Through Time
CNN	Convolutional Neural Network
CPU	Central Processing Unit
FPGA	Field-Programmable Gate Array
LSTM	Long Short-Term Memory
RNN	Recursive Neural Networks
SPSA	Simultaneous Perturbation Stochastic Approximation

Chapter 1

Introduction

Brief overview of the report:

In Chapter 1, I will present a conceptual background to the problem at hand, exposing the background that surrounds it 1.1, the motivation for the desired solution 1.2, its objectives 1.3 and, lastly, the people that will help me achieve them 1.4.

In Chapter 2, the theoretical foundations will be layed 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 2.2.

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

In Chapter **??** I will present some of the work that I have already concluded.

Finally, in Chapter **??**, the work plan for the Dissertation work is detailed as *tasks* in Section **??**, and their temporal arrangement is pictured in a Gantt chart in Section **??**. Furthermore, the description of the Hardware and Software resources that I will be utilizing during the course of my work is done in Section **??**

1.1 Background

Artificial Neural Networks are one of the most popular models 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 among incoming data samples, thus not being suitable to process time-series data, such as audio, video or other kinds of time-varying data streams, where current inputs 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 an large period of time, for the reasons that we will understand in Sections 2.1.3 and 2.1.4.Thus, in 1997 Hochreiter et al. proposed a novel RNN structure [3], 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 [4] (see Section 3.2 for further details), and although it improves the processing time when compared to a naïve software solution, it still lacks the ability to perform on-chip learning, and the learning process is performed offline, in a normal CPU, when it also could be sped up by a dedicated hardware structure.

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 entierely 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, I propose to develop a hardware implementation of an LSTM Network, *with on-chip learning*, improving the performance, capability and flexibility of the existing solution of [4]. Moreover, I also propose to benchmark this solution and try to compare it with the software performance results of [5] and, as a secondary objective, try to use my developed structure to replicate some of the applications portrayed in 3.1. Lastly, I will work to make this structure reconfigurable on the go, and trying to minimize the reconfiguration time.

1.4 People Involved

Besides me, 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 Inteligence.

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.

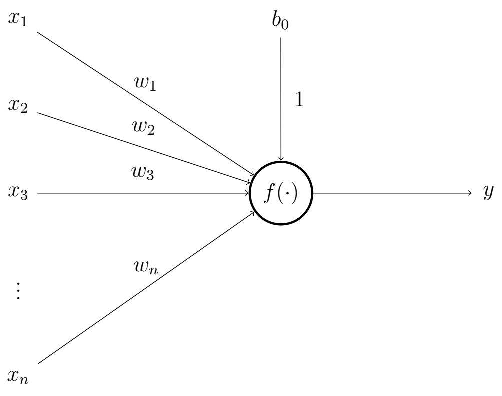
Now that the basic Machine Learning concepts have been presented, I will discuss, henceforth, one of the most important algorithms that address the supervised classification problems, the *Artificial Neural Networks* that will be discussed in Section 2.1.2 as a contextual introduction to the main theme 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

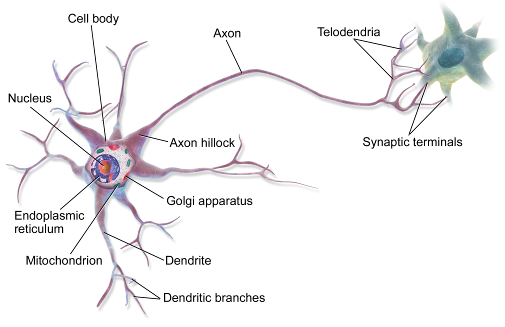
Artificial Neural Networks (ANN) are mathematical structures that, as the name suggests, try to mimic the basic way of how a human brain works. ANN's building blocks, like their biological counterpart, model the high-level behaviour of biological neurons, in the sense that they neglect unnecessary biological 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.

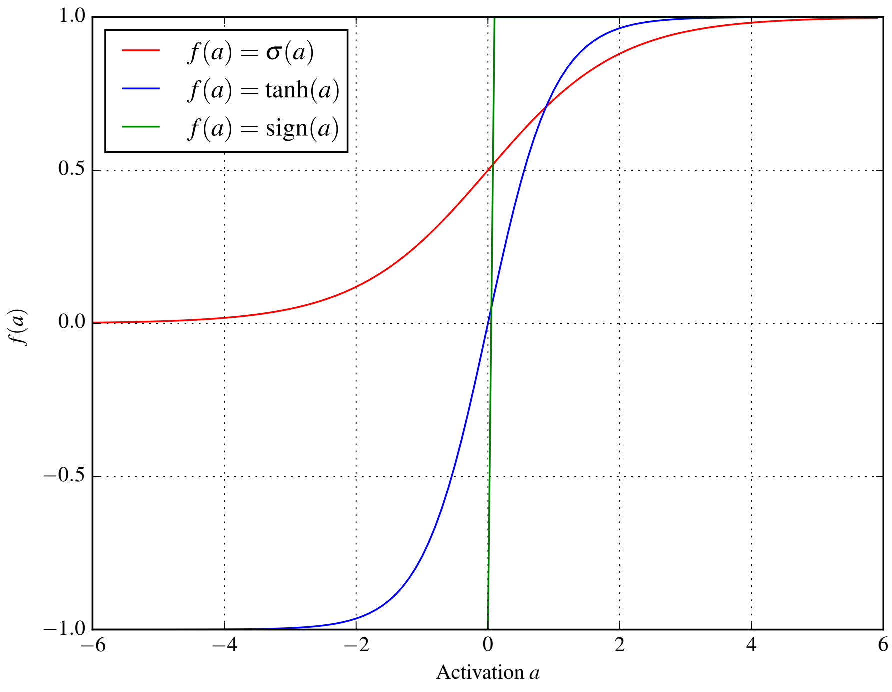


Figure 2.2: Three different activation functions. As you can see, the hyperbolic tangent has the same extreme values as the sign step function, but has a smooth transition between them, which can be interpreted as a *soft decision* in the more ambiguous middle region, reflecting the degree of uncertainty 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.

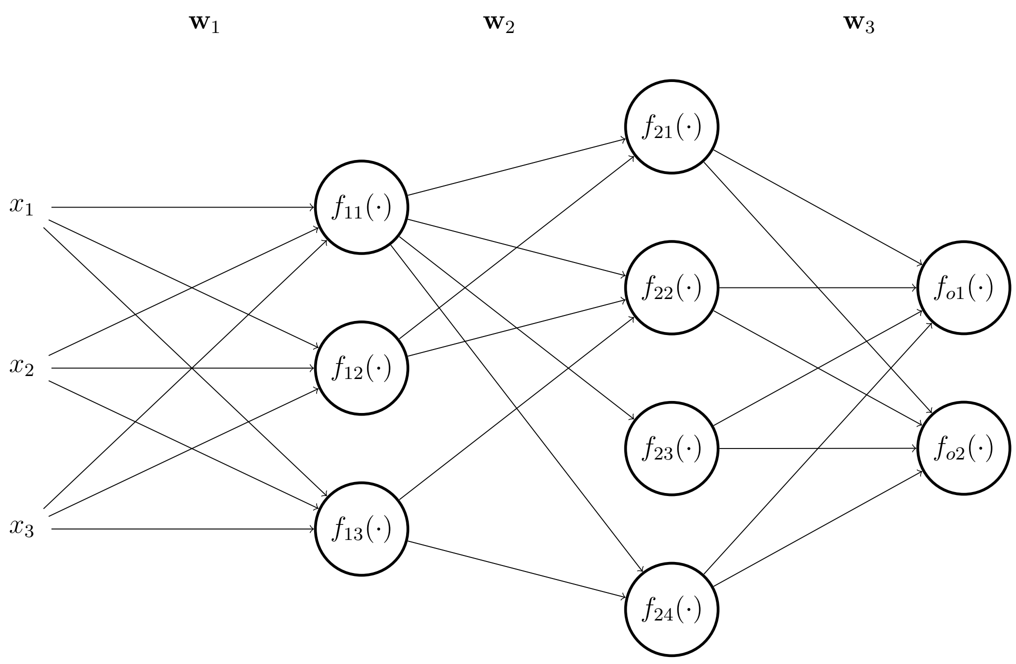


Figure 2.3: A three layer ANN. We have omitted some of the connections in the hidden layer, for simplification purposes. $\mathbf{w}_1$ represents the weight matrix of the input layer, $\mathbf{w}_2$ the weight matrix of the connections between the input layer and the hidden layer, and $\mathbf{w}_3$ 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

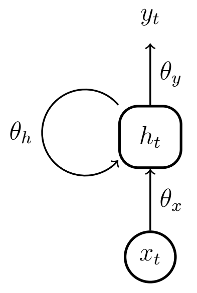
$$w_{ji}^{(\tau+1)} = w_{ji}^{(\tau)} - \eta \frac{\partial E}{\partial w_{ji}} \tag{2.6}$$

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].

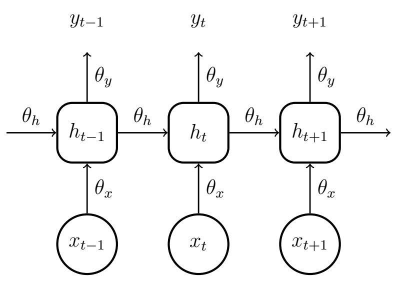
2.1.3 Recurrent Neural Networks

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* [3]. 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

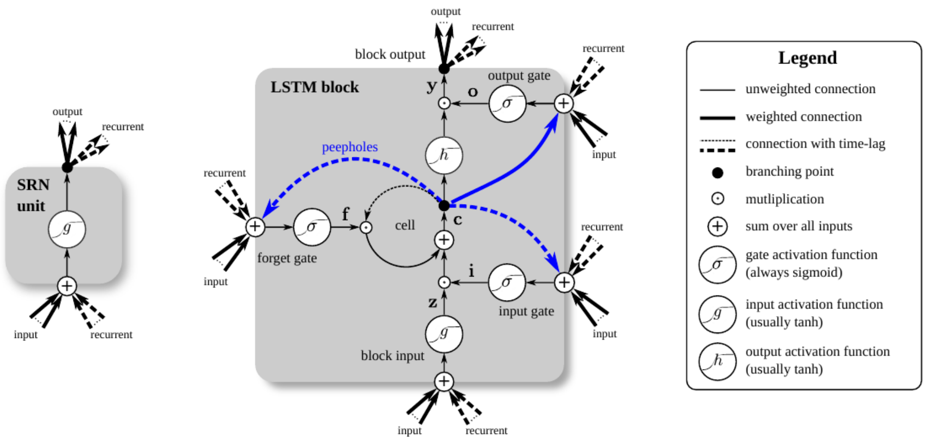


Figure 2.5: A complete LSTM neuron, with all the features as described in [1]. Source: [2]

- **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 perform on-chip learning, 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 the less memory as possible (at the smallest performance cost possible), in order to use the additional memory, for instance, to add additional LSTM cells that can improve the performance of our system. 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] [3] [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 fuction), 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].

2.2 Proposed Solution

Given the previous discussion of how LSTM neurons work and their benefits when compared to other RNN architectures, this work aims to conceive and design an efficient dedicated FPGA implementation with on-chip learning that, surpasses the training/testing time of an equivalent CPU implementation in a general high-level language or of a framework (see Section **??**).

As discussed in Chapter 3, up to now there is only one FPGA implementation of LSTM cells [4], but there is no on-chip learning on the architecture (the weights are learned offline, and loaded in the FPGA before operation, and they are not changed while the FPGA system is operating), and my proposed work will implement an efficient LSTM implementation *with* the ability to learn on-chip, which is an innovative and useful feature for the LSTM architecture on a dedicated hardware implementation. I will achieve that using the SPSA method described in 2.1.4.2, and inspired in the previous work of [13].

This solution will combine the benefits of a dedicated hardware implementation, where the inherent parallelism so obviously visible in the network architecture can be fully exploited something that, predominantly, yields a much higher performance in terms of computation time than a naïve CPU implementation (as in [13] where a $21\times$ speed-up against an ARM CPU was acheived) with the flexibility that a reconfigurable platform such as an FPGA provides.

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, achieveing 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 (November 2015) by Chang et al. [4] in the Computing Research Repository (CoRR). It consists on the proposal of an LSTM cell architecture 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] (the Lua calls are implemented in C, so no performance is lost).

[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.

The implementation itself makes an extensive use of the Multiply-and-Accumulate units (MAC) of the FPGA, which since they are limited in number in the target platform, limits the number of neurons that we can deploy in parallel. The authors report a nearly 75% usage for only 8 LSTM neurons. Although an apparently excessive number, I am left to scrutinize whether the usage of MACs is compulsory to obtain a better performance than a CPU or if it can be discarded, allowing for smaller cells. This way, we can have a layer of more neurons occupying the same area and, hopefully, the same, or even fewer, resources within the FPGA.

3.3 Training Algorithms and Implementation

As stated in 3.2, the work of [4] 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.

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, so it is a very good reference for the design that I will have to implement.

Furthermore, a 2013 article by Tavear et al. [5] 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 restrict it to a smaller use of the FPGA resources, allowing it to accomodate a more complex network. Furthermore, [5] 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. These three conclusions combined indicate that the idea of a hardware LSTM network with on-chip learning using SPSA is viable.

Besides the proposed hardware implementation, it would be advantageous to perform *benchmarking* to see how well it compares with software solutions, and Section 3.1 suggests a handful of examples to test my final solution.

Chapter 4

Proposed Architecture

4.1 Constituent Modules

4.1.1 Finite-precision representation system

Before discussing any details concerned with the actual hardware implementation, I will layout some of the design choices that were made regarding to how we numbers envolved in the calculations will be represented. 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 then 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 [33], which is a good reference for fixed-point arithmetic theory.

4.1.1.1 Precision bitwidth

Since we are dealing with real numbers, and I plan to make use of the DSP48 slices within the FPGA, I chose to use an 18-bit signed fixed-point system, 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. 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 -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),

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

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, our 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 I have attested that the weight values generaly 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 not be very small in order to accomodate the intermediate calculations, avoiding overflow), and so we should choose to have as much precision as possible, so $m = 11$. 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 can be 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 the real numbers to be multiplied, we have that in fixed-point arithmetic, the result c 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 Non-linearity 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, I had to find a suitable way to compute them accurately 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 [34] as a reference on elementary function approximation algorithms, I decided to use **Polynomial Approximations**, 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 does not exceed 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 [35], from 1885, guarantees that there is always a polynomial that can approximate any continuous function f with error $\varepsilon > 0$. Chebyshev also proved [34] that, in a *minimax* polynimal approximation of degree n, the minimum approximation error ε is achieved in at least $n+2$ points, and the sign of approximation error alternates from one interval to the other, and thus the error is not *biased*. 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 *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 $x_i = \frac{a+b}{2} + \frac{(b-a)}{2}\cos\left(\frac{i\pi}{n+1}\right), 0 \le i \le n+1$ –
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 matricial 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 x_is matrix and b is the $f(x_i)$s vector. The Python implementation of this algorithm is presented in Listing 4.1, as follows

```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)

```

```python
# 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)))
```

```python
#print(extremaIndices)
plt.figure(1)
plt.plot(t, pol(t), t, f(t))

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

Listing 4.1: Python script that implements Remez's algorithm

Instead of using a *single* polynomial of higher degree ($n \geq 3$) for the whole domain, I chose to partition the domain of the activation functions in **6 intervals**, and approximate each of those intervals using polynomials of degree $n = 2$. This proved to yield a lower overall approximation error, as expected (the interval on which to perform the approximation is smaller). Also, since both the $\sigma(x)$ and $\tanh(x)$ have horizontal assymptotes 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 assymptote. The resulting *minimax* approximation polynomials yielded by running the code in Listing 4.1 are

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

for the sigmoid function and

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

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 non-linearities 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 –

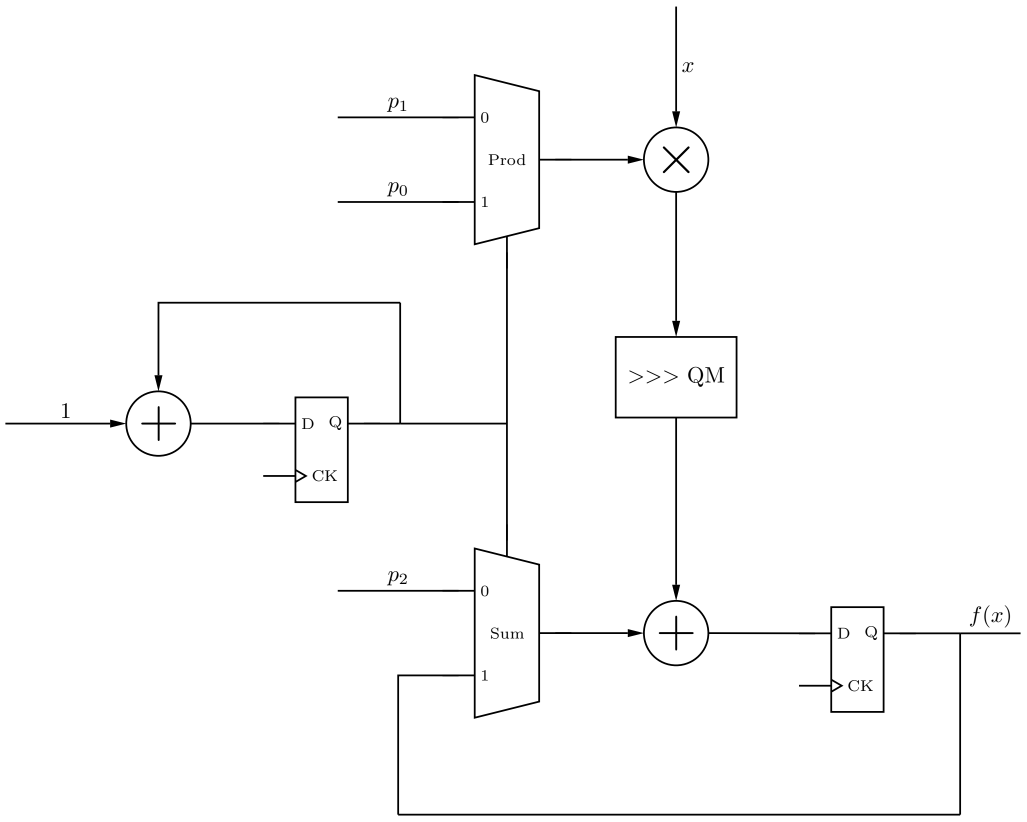


Figure 4.1: Block Diagram of the Non-linearity Calculator using a single multiplication. The multiplexer state is controlled by the flip-flip and sum block, that change state every clock cycle. When reset is applied, the selector is set to zero.

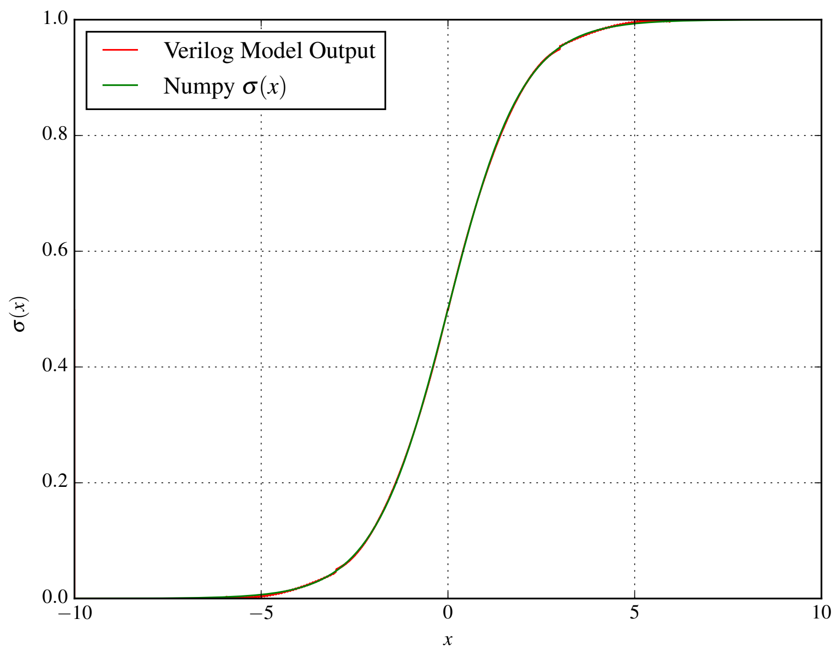
but the DSP slices will be heavily used in the matrix-vector calculators, so it is advisable to save them for that purpose. A simpler approach is to note that by factoring x we get

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

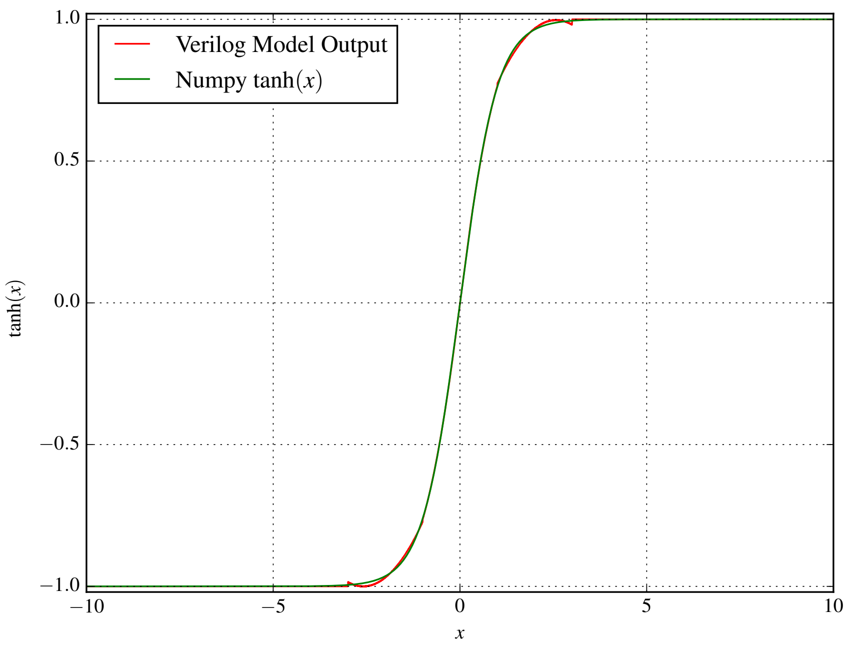
where we can note that this operation can be divided in a two-step procedure of a simultaneous mutiplication of the operand by a constant, and a subsequent addition of another constant: first, by multiplying the operand by p_2 and summing p_1, and then by multiplying the operand by this last result and 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 (I used Numpy as reference) for both 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 a HDL block that implements matrix-vector multiplication, and that block must be parameterized in order to



(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

accomodate 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 lenght N (the number of neurons in the layer), and so $\mathbf{R}_*$ now has size $N \times N$. Plus, if we need a layer with different parameters, either in terms of number of inputs or number of neurons, it 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 $N \times M$ by a vector 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
        $y_i := y_i + A_{ij} \cdot x_j$
    end for
end for
```

This operation has a computational complexity of $O(n^2)$. It can be seen that each of the i-th component of the output vector y can be calculated **in parallel**, each only requiring the corresponding i-th line from the matrix. If we follow 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 run out of resources in the FPGA; therefore, some sort of *resource multiplexing* strategy must be used to ensure the flexibility of the solution to accomodate networks of larger dimensions. The solution I have found for this issue was to *share* the multiplication slice between rows of the matrix: before, each multiplication slice was responsible for producing the i-th element of the output vector 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 we define a parameter $K_G = \frac{\text{Number of rows}}{\text{Number of multipliers}}$ – the number of rows that share the same multiplier – the same 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 $K_G = 2$; 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.

In Figures-4.3 and 4.4 are depicted a diagram of the memory access for the Matrix, and the row multiplication and units within the module, respectively, where I have set $K_G = 4$, for the same matrix and vector sizes as before. Note that in this situation, we would only have 2 multipliers, and 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

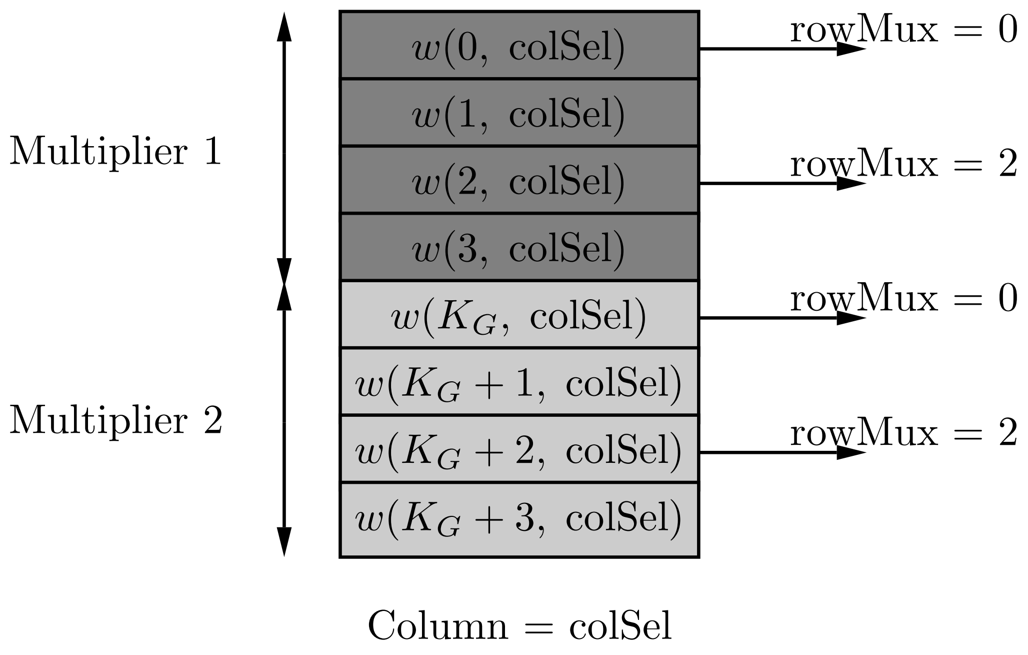


Figure 4.3: A column of the matrix that serves as input to the module. The dark shaded part is for the first multiplier, and the light shaded is for the other, in parallel. The rowMux signal addresses the position within each shaded area

.

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 $K_G - 1$.

In summary, for an $N \times M$ weight matrix, this module outputs the result vector in $K_G \cdot M$ clock cycles.

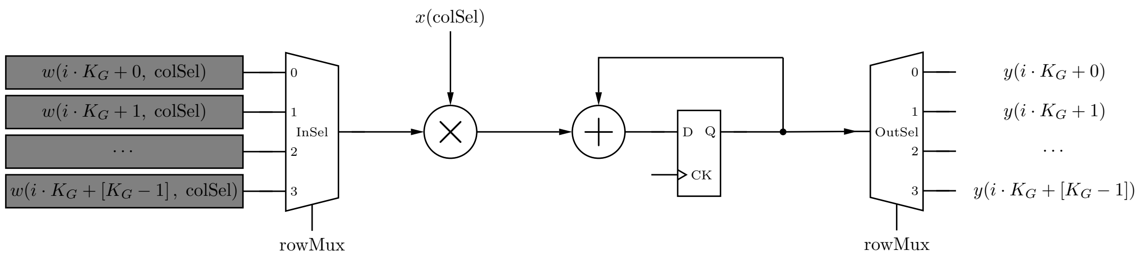

$x(\text{colSel})$
$w(i \cdot K_G + 0,\ \text{colSel})$
$w(i \cdot K_G + 1,\ \text{colSel})$
$\cdots$
$w(i \cdot K_G + [K_G - 1],\ \text{colSel})$
0
1
InSel
2
3
rowMux
D Q
CK
0
1
OutSel
2
3
rowMux
$y(i \cdot K_G + 0)$
$y(i \cdot K_G + 1)$
$\cdots$
$y(i \cdot K_G + [K_G - 1])$

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 $i \cdot K_G + [K_G - 1]$

4.1.4 Dual-port Block RAM

Although the bias weights, being simply a vector, I have chosen to keep them in normal registers and let the synthesis tool decide how to place them in the FPGA, for the weight matrices, the case is different. Since the network size can be big, it is good to make sure that they are placed in the Block RAMs 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 outputs a given column in the `rowOut` output port terminal, at the *negative 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 *negative 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 $\texttt{BITWIDTH} \cdot N$.

The Verilog coding followed the Verilog Coding Guidelines in Xilinx's UG901 that, in page 50, recommends that the `RAM_STYLE` parameter be set to `block`. This can be done by adding the follwing 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 $\texttt{PORT\_BITWIDTH} = \texttt{BITWDITH} \cdot M$ and $\texttt{NCOL} = M$.

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, we can ommit them 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 $\mathbf{x}^{(t)}$
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 $K_G \cdot M$ 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.

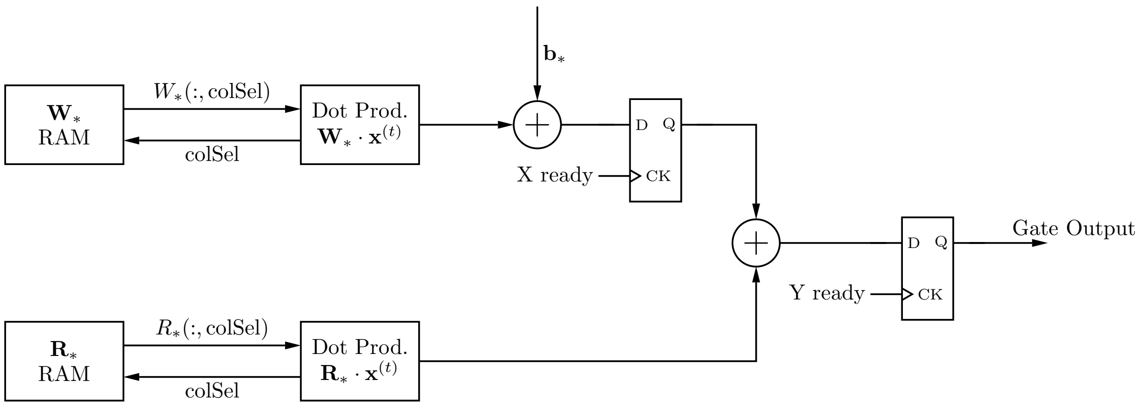


Figure 4.5: Diagram of the hardware block that implements the Gate

4.2 Hardware LSTM Network – Forward Propagation

After detailing the design of each of the constituent modules of the LSTM Network, I will now show how they are interconnected to implement a hardware version of Equations 2.7. This will be performed in Section 4.2.1, where the design decisions and compromises that were taken will be detailed, along with some estimates on the resource usage and calculation time for each of the design iteration versions, the naïve solution (4.2.1.2) and an optimized version of the former (**??**) 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 – they operate only on the current input vector $\mathbf{x}^{(t)}$ and the previous layer output $\mathbf{y}^{(t-1)}$ – we see that 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 Block RAMs for $\mathbf{W}_*$ and $\mathbf{R}_*$, each followed by the respective activation function calculator (detailed in Section 4.1.2). 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 everytime we have a new incoming signal in order to store the last value, which is now $\mathbf{c}^{(t)}$

(Talk about RAM usage)

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

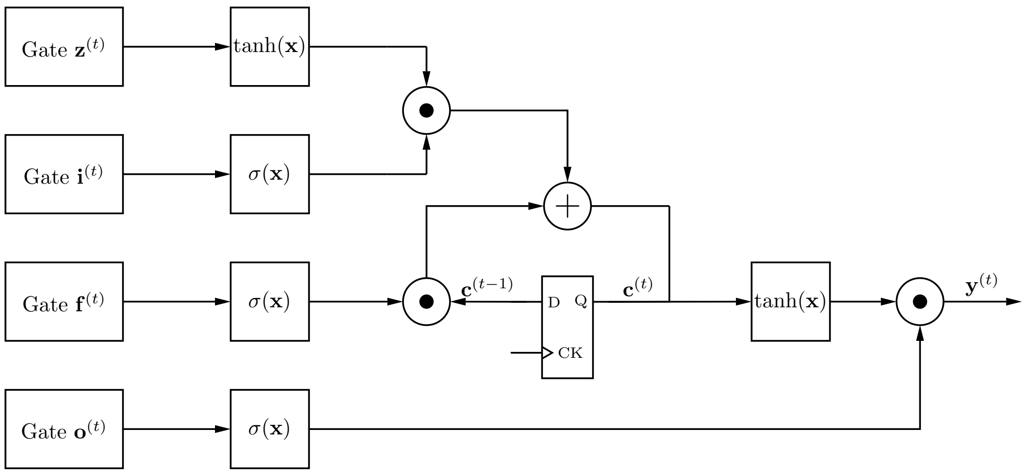


Figure 4.6: Hardware LSTM Network with full level of parallelism

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 elements of the layer, N, 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 $K_G \cdot N$ cycles, the activation function evaluators need always 2, and the elementwise multiplicators need only 1 cycles, and noting that the two elementwise multiplicators that sum to produce $\mathbf{c}^{(t)}$ can work in parallel, the estimated number of clock cycles needed is

$$(N\cdot K_G+2)+(2+1)+(2+1)=8+N\cdot K_G \tag{4.11}$$

4.2.1.2 Shared Elementwise Multiplication Block

In Figure 4.6, we see that, apart from the elementwise multiplicator preceeding the flip-flops, 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 $\sigma(\mathbf{x})$. A clever improvement over the last network architecture is to instead of replicating these tanh-σ-$(\cdot)$wise structures use a *single one* and choose the operand 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 perfect waste of resources. The issue about the elementwise multiplier that precedes the flip-flops can be solved by adding another multiplexer that chooses between the 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.

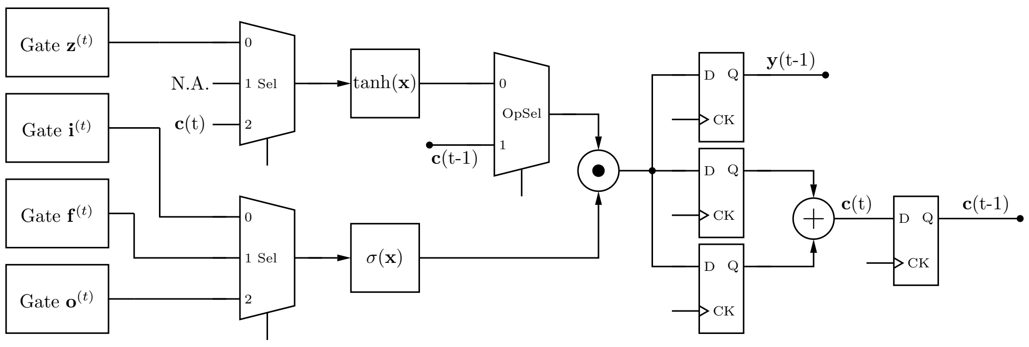


Figure 4.7: Optimised version of 4.6 that shares the elementwise multiplication and the activation function blocks

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 multiplicator (the one that preceeded the flip-flops in the previous design) is the signal $\mathbf{c}^{(t-1)}$, I added another multiplexer before the elementwise multiplication that selects that signal in that particular case, and the output from the $\tanh(\mathbf{x})$ block, otherwise.

The flip-flops on the righthand side of Figure 4.7 are activated by signals generated within the network state machine that enable the appropriate flip-flop, placing the output from the elementwise multiplicator in the correct place. The first activated flip-flop 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 flip-flop to save the other portion of the sum that evaluates to the $\mathbf{c}^{(t)}$ signal. Lastly, the top flip-flop saves the network output $\mathbf{y}^{(t)}$, which in the next incoming 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, 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. The estimated number of clock cycles it takes to produce an output is

$$(N \cdot K_G + 2) + 3 * (2 + 1) = 11 + N \cdot K_G \tag{4.13}$$

which is only 3 clock cycles more than the fully-parallel architecture. For instance, an $N = 32$ neuron network would require 320 DSP slices, while this new architecture would only require 160, only at the expense of 3 more clock cycles.

4.2.2 Timing Diagrams of Operation

4.3 Hardware LSTM Network – SPSA Training

4.3.1 Timing Diagrams of Operation

Chapter 5

Work Plan

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

[4] Andre Xian Ming Chang, Berin Martini, and Eugenio Culurciello. Recurrent neural networks hardware implementation on FPGA. *CoRR*, abs/1511.05552, 2015.

[5] 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.

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

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

[35] K. Weierstrass. Über die analytische darstellbarkeit sogenannter willkürlicher functionen einer reellen veränderlichen. *Sitzungsberichte der Akademie zu Berlin*, pages 633–639, 1885.