FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

# Dissertation Preparation

**José Pedro Castro Fonseca**

WORKING VERSION


Integrated Masters in Electrical and Copmuter Engineering

Supervisor: João Canas Ferreira

Second Supervisor: Ivo Timóteo

February 9, 2016

# Contents

# List of Figures

# List of Tables

# Abreviaturas e Símbolos

| | |
|---|---|
| ANN | Artificial Neural Networks |
| WWW | *World Wide Web* |

# Chapter 1

# Introduction

## 1.1 Background

## 1.2 Objectives

## 1.3 Motivation

# 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 accordingly with 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 it 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 diametrically 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**. Mathematically, if $y^i_\theta = y_\theta(x_i)$ is

the prediction for the input data point $x_i$ with label $t_i$, given the current set of parameters $\theta$, 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 $\theta$), 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 somewhat of 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.

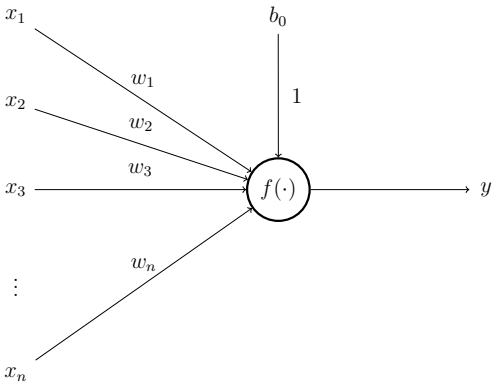
### 2.1.2 Artificial Neural Networks

Artificial Neural Networks (ANN) are mathematical structures that, as the name suggests, try to mimic the Human Brain. 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

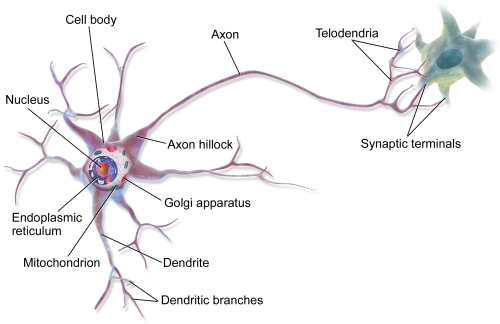
combination of its inputs subject to a *activation function*, i.e. a function that outputs a decision value depending on its inputs. Mathematically, we have

$$y = f(\mathbf{w}^T\mathbf{x} + b_0) \tag{2.2}$$

where $\mathbf{w} = [w_1\ w_2\ w_3\ \dots\ w_n]$ is the input weight vector, $\mathbf{x} = [x_1\ x_2\ x_3\ \dots\ x_n]$ the input vector, $b_0$ is the bias factor and $f(\cdot)$ is the chosen activation function. Furthermore, we call the scalar quantity $a = \mathbf{w}^T\mathbf{x} + b_0$ 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.

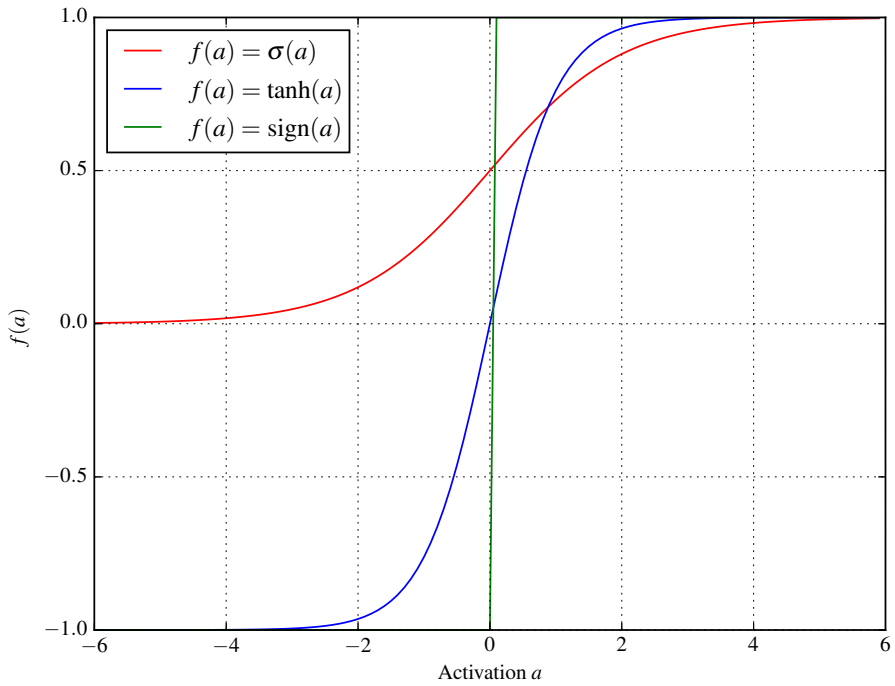



Figure 2.2: Three different activation functions. As you can see, the hyperbolic tangent has the same extreme value as the sign step function, but has a smooth transition between them, which can be interpreted as a *soft decision* in the more ambiguous middle region, reflecting the degree of uncertainty 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 *initial* layer, the *hidden* layer (in the middle) and the *output* layer. Figure 2.3 suggests a possible structure for a 3 layer ANN.

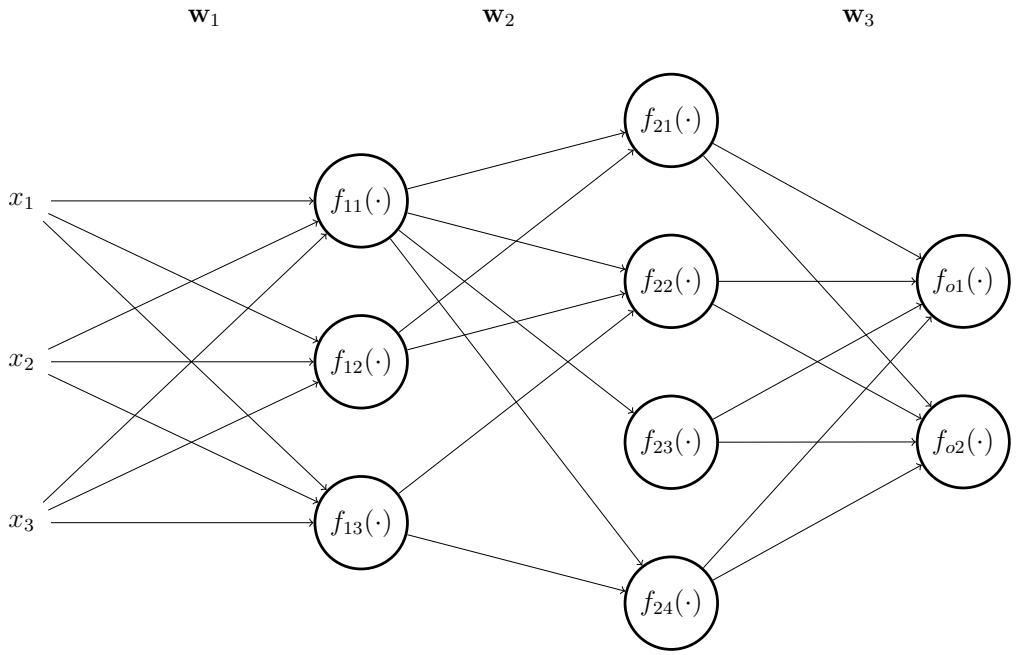



Figure 2.3: A three layer ANN. We have omitted some of the connections in the hidden layer, for simplification purposes. $\mathbf{w}_1$ represents the weight matrix of the input layer, $\mathbf{w}_2$ the weight matrix of the connections between the input layer and the hidden layer, and $\mathbf{w}_3$ 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, is 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, the perform the **backpropagation** step, where we calculate the errors in each of the nodes, but now from the output to the input: the weights are updated and optimized using an iterative method called *Gradient Descent*, where if $\tau$ 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 $\eta$ 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 on $w_{ji}$, 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, $\delta_j$, 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 $\delta_j$ 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 [3, 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 no information about the correlation between themselves and the data points that preceded them did not influence the training step. They were regarded as if no temporal relationship existed, and therefore each data point is conditionally independent of any other. This is obviously not necessarily truth, and in fact there are many cases, where the correlation between data points is high for those closely spaced in time, in which it is actually completely false, 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 **??** 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.

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 backpropagation procedure discussed for the ANNs, but now taking into account the unfolding of the

network through a fixed training epoch $T$ like Figure **??**. 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 **Simulatneous Perturbation Stochastic Approximation**.

Even though RNNs outperform static ANNs in sequence recognition problems [4], 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 [5], and therefore this memory only retains short-term dependencies. This is because of the so-called **Vanishing Gradients Problem** [6, 5], 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* [7]. This section explains the main idea of this approach 2.1.4.1, and also how it is trained2.1.4.2, serving as a support basis for the work of this thesis. Although originally formulated in 1997, its formulation has been incrementally updated in [8] and [9], 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 structure [2] last year, and this will the basis of this short theoretical presenation, 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 2.4. 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 ommited from the picture, all the gates are biased, as is suggested in Equations **??**. The main components, their role and relevance, are explained as follows

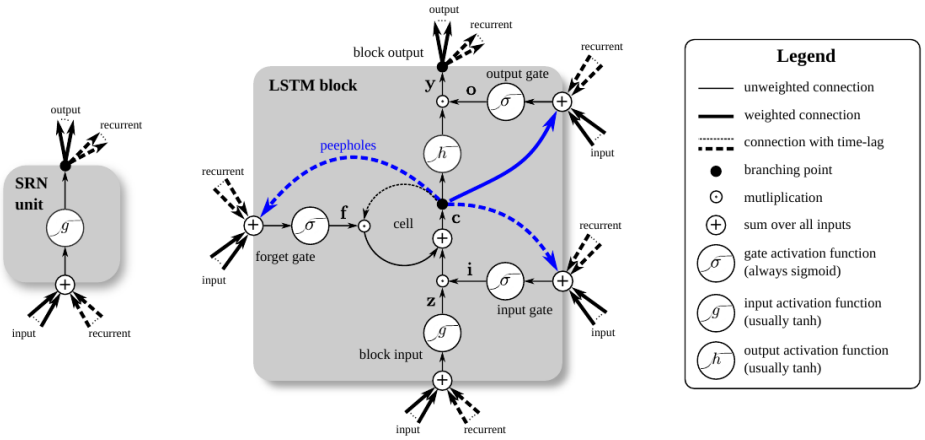



Figure 2.4: A complete LSTM neuron, with all the features as described in [1]. Source: [2]

- **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 any**, 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' 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

$$\mathbf{z}^{(t)} = g(\mathbf{W}_z\mathbf{x}^{(t)} + \mathbf{R}_z\mathbf{y}^{(t-1)} + \mathbf{b}_z) \tag{2.7}$$
$$\mathbf{i}^{(t)} = \sigma(\mathbf{W}_i\mathbf{x}^{(t)} + \mathbf{R}_i\mathbf{y}^{(t-1)} + \mathbf{p}_i \odot \mathbf{c}^{(t-1)} + \mathbf{b}_i) \tag{2.8}$$
$$\mathbf{f}^{(t)} = \sigma(\mathbf{W}_f\mathbf{x}^{(t)} + \mathbf{R}_f\mathbf{y}^{(t-1)} + \mathbf{p}_f \odot \mathbf{c}^{(t-1)} + \mathbf{b}_f) \tag{2.9}$$
$$\mathbf{c}^{(t)} = \mathbf{i}^{(t)} \odot \mathbf{z}^{(t)} + \mathbf{f}^{(t)} \odot \mathbf{c}^{(t-1)} \tag{2.10}$$
$$\mathbf{o}^{(t)} = \sigma(\mathbf{W}_o\mathbf{x}^{(t)} + \mathbf{R}_o\mathbf{y}^{(t-1)} + \mathbf{p}_o \odot \mathbf{c}^{(t)} + \mathbf{b}_o) \tag{2.11}$$
$$\mathbf{c}^{(t)} = \mathbf{o}^{(t)} \odot h(\mathbf{z}^{(t)}) \tag{2.12}$$

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 nowadays 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] [7] [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) [10]. The weight update for the $i$-th

weight at the time step $t$ is given by

$$\Delta w_i^{(t)} = \frac{J(\mathbf{w}^{(t)} + \beta \mathbf{s}^{(t)}) - J(\mathbf{w}^{(t)})}{\beta s_i^{(t)}} \tag{2.13}$$

where $\beta$ 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 $\beta$, 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* is concerned, we have

$$w_i^{(t+1)} = w_i^{(t)} - \eta \Delta w_i^{(t)} \tag{2.14}$$

where $\eta$ is the learning rate, and $\Delta w_i^{(t)}$ is the update for the $i$-th weight evaluated in 2.13. According to the analysis performed in [11], performing a second-order Taylor expansion at $\mathbf{w} = \mathbf{w}^{(t)}$, and taking the expected value of the equation, we get that

$$E(\Delta w_i^{(t)}) = \frac{\partial J(\mathbf{w}^{(t)})}{\partial w_i^{(t)}} \tag{2.15}$$

that is, 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 hipothetical 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 [11].

## 2.2 Proposed Solution

Given the previous discusson of how LSTM neurons work, their benefits when compared to other RNN architectures, this work aims to conceive an efficient dedicated FPGA implementation with on-chip learning that, hopefully, surpasses the training/testing time of an equivalent CPU implementation in a general high-level language or from a framework.

As discussed in Chapter 3, hitherto there is only one FPGA implementation of LSTM cells [12], 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 [11].

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 [11] where a 21× speed-up against an ARM CPU was acheived.

# 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 is useful by itself, 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)

## 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. [12] in the Computing Research Repository (CoRR). Is 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, written in Lua using the Torch7 framework (the Lua calls are implemented in C, so no performance is lost).

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 less, resources within the FPGA.

## 3.3 Training Algorithms and Implementation

## 3.4 Final Overview

# Chapter 4

# Work Plan

## 4.1 Main Tasks

## 4.2 Approach

## 4.3 Tentative Scheduling

## 4.4 Software and Hardware Resources to be used

# Chapter 5

# Early Work

# Appendix A

# Loren Ipsum

Depois das conclusões e antes das referências bibliográficas, apresenta-se neste anexo numerado o texto usado para preencher a dissertação.

## A.1 O que é o *Loren Ipsum*?

## A.2 De onde Vem o Loren?

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

[4] 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).

[5] Sepp Hochreiter, Yoshua Bengio, Paolo Frasconi, and Jürgen Schmidhuber. Gradient flow in recurrent nets: the difficulty of learning long-term dependencies. 2001.

[6] 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. Gradient based learning algorithms;Information latching;Input/output sequences;Learning algorithms;Parametric dynamical system;Recurrent neural network training;Temporal contingencies;.

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

[8] F.A. Gers, J. Schmidhuber, and F. Cummins. Learning to forget: continual prediction with lstm. *Neural Computation*, 12(10):2451 – 71, 2000. LSTM;learning algorithms;recurrent neural networks;LSTM networks;continual input streams;forget gate;.

[9] F.A. Gers and J. Schmidhuber. Recurrent nets that time and count. volume vol.3, pages 189 – 94, Los Alamitos, CA, USA, 2000. recurrent neural networks;RNN;timing;counting;time intervals;sequential tasks;motor control;rhythm detection;long short-term memory;LSTM;peephole connections;internal cells;multiplicative gates;discrete time steps;stable sequences;highly-nonlinear precisely-timed spike sequences;.

[10] J.C. Spall. Adaptive stochastic approximation by the simultaneous perturbation method. volume vol.4, pages 3872 – 9, Piscataway, NJ, USA, 1998. stochastic approximation;loss functions;stochastic search;Newton-Raphson algorithm;Hessian matrix;iterative method;optimization;root-finding;parameter estimation;adaptive estimation;.

[11] 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. perturbation learning rule;recurrent neural network;FPGA;dynamic information processing;feedforward neural network;recursive learning scheme;correlation learning;analog learning;oscillatory solution;hardware implementation;Hopfield neural network;field-programmable gate array;.

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