A Hardware Accelerated Low Power DSP for Recurrent Neural Networks

ALLAN ANDERSEN
ILAYDA YAMAN
DEPARTMENT OF ELECTRICAL AND INFORMATION TECHNOLOGY
FACULTY OF ENGINEERING | LTH | LUND UNIVERSITY

A Hardware Accelerated Low Power DSP for Recurrent Neural Networks

Master thesis

Allan Andersen
macadk@gmail.com

Ilayda Yaman
ilaydayaman@gmail.com

Department of Electrical and Information Technology
Faculty of Engineering | LTH | Lund University
Lund | Sweden
June 2020

Acknowledgments

We would like to thank our supervisor Joachim Rodrigues for all the support and feedback we have received during this work, as well as the tools that were made available to us. We are very grateful that it was made possible, to work with processor design in Tensilica. Also, our second supervisor, Ph.D student Lucas Ferreira, we thank you for always taking your time for us. Your assistance has been very appreciated by us. Lastly, we would like to thank our families and friends for the support during this period.

Allan Andersen & Ilayda Yaman
June 2020

Abstract

Recurrent neural networks (RNNs) have become a dominating player for processing of sequential data such as speech and audio. The reason for this, is the high accuracy that can be achieved with the more complex variants, such as the gated recurrent unit (GRU). This makes them very attractive in speech recognition systems for digital assistance and voice control applications. However, a high power consumption and the large amount of memory required for these networks, make them less suitable for battery powered devices. In this work, we have designed a system on a chip (SoC) for efficient processing of GRU networks, that consists of an optimized digital signal processor (DSP) integrated with a hardware accelerator. To deal with the large memory requirements and high power consumption, several optimization techniques have been applied. A 75% reduction is achieved for the required memory, while the system can process real-time speech data with an energy consumption of 7.79 μJ per classification. In 28nm CMOS technology the area is 0.686 mm^2. The design is programmable and scalable, which allows for execution of different network sizes.

Popular Science Summary

New advancements in AI techniques and designs come into light everyday but there are serious limitations that should be addressed before they can be a part of the everyday life. In March 2016, Lee Sedol, who is a professional top-ranking Go player with 18 world championship titles, lost a match to an AI-based computer program called, AlphaGo. This victory (or defeat depending on how you look at it) was a game changer and milestone for the AI community in many ways. On the contrary, according to estimations, the AlphaGo consumes 1 megawatt power whereas the human brain consumes only 20 watts[1], which means the human brain is 50,000 times more efficient than the famous computer program.

Another area where AI is becoming a main player is recognition tasks such as image or speech recognition. Speech recognition systems are already commonly used in daily life and many people have interacted with them through "Apple Siri", "Google Home" or other kind of speech recognition systems. Many of these systems are using machine learning methods in order to classify the words spoken by humans. Machine learning methods focus on developing computer programs that can learn on their own and deep neural networks are a subset of machine learning. Deep neural networks can be specialized in accomplishing tasks such as detecting the keywords in a speech. Within deep learning, there are recurrent neural networks, which are more appealing than the other techniques because of the high accuracy it performs on speech recognition tasks. Yet, the high power consumption and the huge amount of data necessary in order to complete the calculations are restricting the usage of this technique. Many platforms that are used today just do these calculations in the cloud, which means the data on the device will be sent via the internet to large data centers where it is processed. This

has many negative results such as extra power consumption, latency, cloud dependency and concerns about the security.

In our thesis, we explore different design techniques in order to achieve a low power, high efficiency and scalable design for speech recognition. The reference model we use is based on gated recurrent unit (GRU) which is an advanced recurrent neural network and it is used to detect a keyword that is spoken such as "Hey Siri" or "Okay Google". In order to implement a scalable yet highly efficient design, we optimized a Digital Signal Processor based on an Xtensa processor by Cadence Design Systems and accelerated the computation and optimized the memory usage with a hardware accelerator.

List of Acronyms

AI	Artificial Intelligence
ASIC	Application Specific Integrated Circuit
API	Application Programming Interface
ALU	Arithmetic Logic Unit
AXI	Advanced eXtensible Interface
BPTT	Backpropagation Through Time
CMOS	Complementary metal–oxide–semiconductor
CNN	Convolutional Neural Network
CPU	Central Processing Unit
CU	Control Unit
DNN	Deep Neural Network
DRAM	Dynamic Random Access Memory
DSP	Digital Signal Processing
FC	Fully Connected
FIFO	First In First Out
GPR	General Purpose Register
GPU	Graphics Processing Unit

GRU	Gated Recurrent Unit
HMM	Hidden Markov Model
IoT	Internet of Things
KWS	Keyword Spotting System
LSB	Least Significant Bit
LSTM	Long Short Term Memory
LUT	Lookup Table
LVT	Low Voltage Threshold
MFCC	Mel-Frequency Cepstral Coefficient
MLP	Multilayer Perceptron
MSB	Most Significant Bit
NMOS	N-type metal–oxide–semiconductor
PMOS	P-type metal–oxide–semiconductor
RISC	Reduced Instruction Set Computing
RAM	Random Access Memory
RALUT	Range Addressable Lookup Table
RNN	Recurrent neural network
ROM	Read Only Memory
RTL	Register Transfer Level
SoC	System on a Chip
SRAM	Static Random Access Memory
TPU	Tensor Processing Unit

Contents

List of Figures

List of Tables

Chapter 1

Introduction

With the technology advancements in computer hardware, deep neural networks (DNNs) have become widely used in many applications, e.g., within the areas of speech recognition, computer vision and robotics where DNNs are used to achieve performance that in many cases exceeds human accuracy [2]. The cost for the superior performance is high computational complexity, which is why graphics processing units (GPUs) have been the natural choice in many DNN applications [2]. While GPUs are very useful for the computation of extensive tasks, their power consumption is far beyond what most battery powered devices can provide [3].

For many DNN systems behind well-known applications like the trigger word detection systems Siri, Hey Google and Alexa, the data processing takes place in the cloud. Recent studies have shown that due to latency, cloud dependency, security and privacy [4], it would be more efficient for many of the applications to perform the DNN processing directly on the edge, that is the end-user devices and Internet of Things (IoT) [2]. A challenge in implementing the technology in devices on the edge, is that many of these devices typically have limited resources in terms of memory, power supply and computational power [2]. As a result, there has been an increased demand for efficient, adaptive and scalable hardware for DNNs that can perform on such devices [2][5].

Two important DNN types are feedforward networks and recurrent networks. Only within the last few years, hardware acceleration for recurrent networks has started to draw more attention in research [2].

In order to address the demand for efficient hardware to execute recurrent

networks on the edge, this master's thesis seeks to develop a scalable and low-power digital signal processor (DSP) together with an accelerator, that is optimized for the inference of recurrent networks. A DSP is programmable, providing the opportunity for adaptation to a fast evolving area. An efficient accelerator can be exploited for reducing the power consumption.

1 Scope of Thesis

The goal of this master's thesis work is to design a low power and memory efficient DSP and a hardware accelerator, for inference of recurrent neural networks (RNNs) on the edge. Two popular forms of RNNs are the long short-term memory (LSTM) and the gated recurrent unit (GRU). The LSTM includes more parameters than the GRU, while a similar accuracy can be achieved with the GRU in many cases [6]. A recent study has shown that the GRU can outperform the LSTM when used in speech recognition systems [7]. In that scope, the DSP is designed for RNNs based on the GRU with focus on speech recognition system as the application.

In order to achieve a low power and memory efficient design, implementation of optimization techniques plays a significant role. This leads to the following question that is addressed in this thesis work.

- How and which type of optimization techniques can be applied to achieve a low power and memory efficient design, that involves a DSP and a hardware accelerator?

2 Related Work

With the extensive computational intensity required by deep neural network algorithms, many powerful hardware accelerator designs have been proposed both in academia and from the industry [5][8][9]. Most of them focuses on feedforward and convolutional neural networks, and little attention has been given to RNNs. Some of the interesting work where the acceleration of RNNs have been studied are presented below.

One example from industry is Google's tensor processing unit (TPU) that can accelerate both MLPs, CNNs and RNNs. However, with a power consumption of 40 watts, this unit is not fitted for battery powered devices and IoTs, as the battery would quickly be drained and heating would be a problem.

Verhelst M. et al. [5] proposes a programmable LSTM accelerator called Laika which can be used for always-on keyword spotting systems(KWS) [1]. One remarkable feature of this design, is that the power consumption is only 5μW while it achieves 60 nJ per inference. It is also doing processing in real time, which is defined as computing a feature vector with 39 features per 32 ms in this paper. On the other hand, with 32 kilobytes of memory only small to medium sized LSTM networks can be realized in Laika and it does not allow high flexibility to the system such as being able to add layers such as a batch normalization layer, before or after LSTM layers. In some RNN networks, these modifications can be essential and have an impact on achieving high accuracy.

The Deep neural network processing unit (DNPU) [8] is another example from the academia which implements an energy efficient deep learning processor for mobile platforms. The multicore architecture achieves 8.1 TOP-S/W energy efficiency using 34.6 mW power for both CNNs and RNNs. DNPU is very flexible but not specific for speech recognition applications and only 10 kilobytes out of 290 kilobytes of SRAM, is dedicated to RNN and MLP networks, which does not allow for inference of larger RNN models. A trade-off for the higher level of flexibility and larger memory space, is an increased power consumption. With 34.6 mW, the DNPU cannot compete with a dedicated LSTM accelerator like Laika and it is less attractive for mobile platforms where a low power consumption is important.

3 Thesis Organization and Outline

With the objective for the thesis defined, the steps taken to reach the goal are outlined below.

1. Literature research and establishing the design specifications with respect to commonly used speech recognition systems.

2. Decide on pre-trained RNN model that is based on GRUs. For extraction of model parameters and to be used as a reference model.

3. Creation of the reference RNN model in C programming language, for execution and profiling on a state-of-the-art DSP, to identify potential bottlenecks.

[1] Always-on keyword spotting systems (KWS) have the same functionalities as trigger word detection systems

4. Development of the fixed-point RNN model in MATLAB for optimization and verification purposes.

5. Design of optimized DSP using Cadence Tensilica and optimization of the C-code.

6. Creation of of hardware accelerator in high definition language.

7. Verification of DSP and accelerator by simulations.

8. Synthesis and Place and Route of the hardware accelerator.

The work of this thesis is detailed over eight chapters. In chapter 2, the theory behind deep learning, RNNs and the GRU is explained. Chapter 3, analyzes the aspects considered in defining the hardware requirements and specifications. Optimization techniques are discussed in chapter 4. The hardware implementation of the processor is explained in chapter 5. The designed hardware accelerator is described step by step in chapter 6. In chapter 7 are the system verification and implementation results presented and discussed. Lastly, the 8th chapter concludes this thesis and ideas for future work are provided.

Chapter 2

Deep Learning - A Brain Inspired Approach

Artificial intelligence (AI), is the science and engineering of making intelligent machines, especially intelligent computer programs, according to John McCarthy, a computer scientist who is considered to be one of the founders of AI.

An area of AI is neural networks that are designed to perform tasks by learning from examples. They are inspired by the way it is believed the human brain works and with the advancements in technology, they have become a key player in many applications [2]. Within the area of neural networks are deep neural networks (DNNs) also referred to as deep learning. DNNs have one or more hidden layers in between the input and output layer and are able to learn more complex tasks [10].

In the brain the neuron is the main computational element [2]. As illustrated in Figure 2.1, the neuron is essentially performing a computation on input signals called dendrites and the output of the neuron is called an axon. The axon branches out to become dendrites for other neurons. In the synapse, where an axon becomes a dendrite, the signal is scaled with a certain parameter or weight (W). The output of the neuron is defined as [2]

$$y_j = f\left(\sum_{i=1} W_i x_i + b\right), \tag{2.1}$$

where $b$is a bias and $f(\cdot)$ is a non-linear function, which allows for modelling of non-linear systems. The bias is used to add a threshold i.e. it allows for

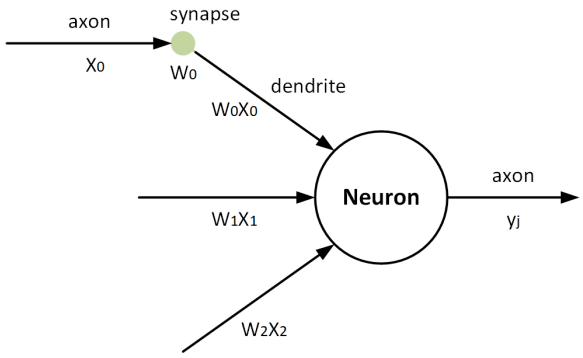

Figure 2.1: Neuron as a computational element. Figure adapted from [2].

shifting the values.

Common used non-linear functions includes sigmoid (2.2), hyperbolic tangent (2.3) and the rectified linear unit (ReLU) (2.4)[1] [2] given as

$$\sigma(x) = \frac{1}{1 + e^{-x}}, \tag{2.2}$$

$$\phi_h(x) = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}}, \tag{2.3}$$

$$ReLU(x) = max\big(0, x\big). \tag{2.4}$$

A simple neural network that is inspired by the brain, is the multilayer perceptron (MLP), also known as the fully connected layer (FC Layer), which is a feedforward neural network. An example of this network is provided in Figure 2.2. As seen, there is only one hidden layer, however most models include many hidden layers. The computations are performed in a sequence, where the output of a previous layer is the input to its following layer. By applying equation (2.1), the network output y, is computed as follows:

$$\begin{aligned} y &= f(U_1 h_1 + U_2 h_2 + U_3 h_3 + b), \\ h_1 &= f(W_{11} x_1 + W_{21} x_2 + W_{31} x_3 + b_1), \\ h_2 &= f(W_{12} x_1 + W_{22} x_2 + W_{32} x_3 + b_2), \\ h_3 &= f(W_{13} x_1 + W_{23} x_2 + W_{33} x_3 + b_3). \end{aligned}$$

[1] A graphical representation for the three functions is found in Appendix B.

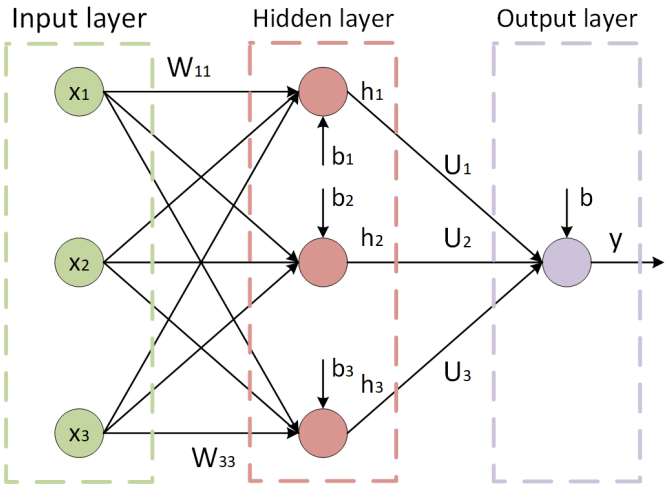

Figure 2.2: Multilayer perceptron.

Feedforward neural networks or MLPs are efficient tools to do e.g. classifications of items [11]. By increasing the number of hidden layers and the neurons within, more complex tasks can be carried out. What is essential to note about this type of network, is that for the input data no information about possible dependencies in time are captured, meaning that an output computed for a given input, is not affected by previous provided inputs to the network. Another type of network is the recurrent neural network (RNN), which is able to capture these short and long term dependencies in time in the data. In the next section this type of network will be presented.

1 Recurrent Neural Network

A recurrent neural network is said to have internal memory by which it can capture dependencies in a sequence of data. A feedback path is incorporated to the network so the output is not only dependent on the current input but also on the previous value of the network. A simple RNN with a feedback path in the hidden layer is illustrated in Figure 2.3a.

As with the MLP, the network output $y(t)$ is computed at each time step t. A significant difference with RNN is the property of the hidden layer which includes a feedback path. This feedback path causes the output of the

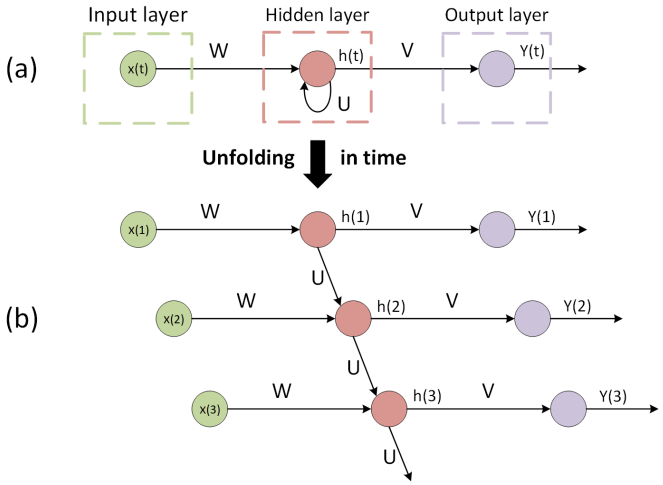

Figure 2.3: Simple RNN model (a) and the correspondent model unfolded in time (b).

hidden layer to become a non-linear function of not only the current input *x(t)*, but also on the previous value *h(t-1)* of the hidden layer, as shown in

$$h(t) = f\Big(Wx(t) + Uh(t-1) + b_h\Big), \tag{2.5}$$

where f is the non-linear function. The impact the previous value *h(t-1)* will have on the next value, is determined by the parameter U. Looking at the network unfolded in time, as illustrated above in part b of Figure 2.3, the network output *y(t)* is computed for each time step as:

$$\begin{aligned}
y(1) &= f\Big(Vh(1) + b_f\Big) = f\Big(V\, f\Big(Wx(1) + Uh(0) + b_h\Big) + b_f\Big),\\
y(2) &= f\Big(Vh(2) + b_f\Big) = f\Big(V\, f\Big(Wx(2) + Uh(1) + b_h\Big) + b_f\Big),\\
y(3) &= f\Big(Vh(3) + b_f\Big) = f\Big(V\, f\Big(Wx(3) + Uh(2) + b_h\Big) + b_f\Big).
\end{aligned}$$

As the duration for input sequences lengthens, the simple RNN is not successful in capturing the data dependencies, which leads to a drop in correct predictions. This is an issue known as the vanishing gradient problem that occurs during the training of the network, and it becomes more and more apparent as the sequences grow with time, due to the large compositions of functions. The vanishing gradient problem is discussed in more details in the next section.

2 Training of Neural Networks

Training of neural networks is important in order for a network to do the task it is designed for, while keeping the difference between the network output and what is expected at a minimum. Training a neural network means determining the network parameters that minimizes this difference, which is typically defined by an error function. A common approach is to first define the error function and then minimize it by using gradient descent[2] and back-propagate an error signal through the layers of the network, in order to update the network parameters.

To train a simple RNN, as presented in the previous section, an input data set and a target sequence are required and the error function must be defined. By using gradient descent, the derivative (or gradient) of the error function is computed with respect to each of the weights V, U and W (see in Figure 2.3) as,

$$\frac{\partial E}{\partial V} = \sum_t \frac{\partial E_t}{\partial V},$$
$$\frac{\partial E}{\partial U} = \sum_t \frac{\partial E_t}{\partial U},$$
$$\frac{\partial E}{\partial W} = \sum_t \frac{\partial E_t}{\partial W}.$$

For V, the derivative is determined using the chain rule as,

$$\frac{\partial E_t}{\partial V} = \frac{\partial E_t}{\partial y(t)} \frac{\partial y(t)}{\partial V}.$$

For U and W, the derivative is determined in the same way, which for U is,

$$\frac{\partial E_t}{\partial U} = \frac{\partial E_t}{\partial y(t)} \frac{\partial y(t)}{\partial h(t)} \frac{\partial h(t)}{\partial U}.$$

As seen in the unfolded network under part b in Figure 2.3, the term *h(t-1)* depends on U from the current time step and back to the first time step. The derivative of U is therefore the sum of all of the gradients,

$$\frac{\partial E_t}{\partial U} = \sum_{k=0}^{t} \frac{\partial E_t}{\partial y(t)} \frac{\partial y(t)}{\partial h(t)} \frac{\partial h(t)}{\partial h(k)} \frac{\partial h(k)}{\partial U}. \tag{2.6}$$

[2]Gradient descent is an iterative algorithm for finding the minimum of a differentiable function.

This method is called Back-propagation Through Time and is a common way of training recurrent neural networks. However, for sequences with long duration, the composition of functions evolves and consequently the longer chain rule of derivatives, increasing the risk of the gradients becoming very small or vanishing. Looking at equation 2.6, the derivative term $\partial h(t)/\partial h(k)$ is a chain rule in itself and involves many multiplications of derivatives of activation functions as the sequence duration increases. For example, for t=3,

$$\frac{\partial h(3)}{\partial h(0)} = \frac{\partial h(3)}{\partial h(2)} \frac{\partial h(2)}{\partial h(1)} \frac{\partial h(1)}{\partial h(0)}.$$

There is a high risk of multiplying small numbers, and the result of this is referred to as the vanishing gradient problem. One way to deal with this problem is to implement a more advanced hidden layer, like the gated recurrent unit, which will be introduced in the next section.

3 Gated Recurrent Unit

The gated recurrent unit was first introduced by Kyunghyun Cho in 2014 [12]. It was motivated by the long short term memory (LSTM) unit, originally proposed by Graves and Schmidhuber [13]. An advantage with the GRU is that it is less complex to compute and requires less parameters [12]. The central idea behind the LSTM and the GRU units is a memory cell that can maintain its state over time and gating units for controlling the data flow in and out of the memory cell [14]. This approach helps to overcome the vanishing gradient problem.

Looking at the GRU, the three main components are referred to as the update gating unit, reset gating unit and the memory cell as illustrated in Figure 2.4. For readability *x(t)*, *h(t)* and *h(t-1)* is denoted as x_t, h_t and $h_{t\text{-}1}$. The same notation is used for u, r and $\hat{h}$.

The output *h(t)* of the GRU unit is used both as a feedback for the hidden layer itself and as an input for the next layer in the network.

The memory cell stores information used to capture long and short term dependencies in the input sequence. The output of the cell $\hat{\mathrm{h}}(\mathrm{t})$, is a potential candidate to become the new *h(t)*, which is the GRU output. Whether it is used or not is determined by the update gating unit. A hyperbolic tangent function is utilized for the activation function. The memory cell is defined as in [12]

$$\hat{h}_t = \phi_h\Big(\big[\mathbf{W^c x_t}\big] + \big[\mathbf{U^c}\big(\mathbf{r_t} \odot \mathbf{h_{t-1}}\big)\big] + \mathbf{bias^c}\Big). \tag{2.7}$$

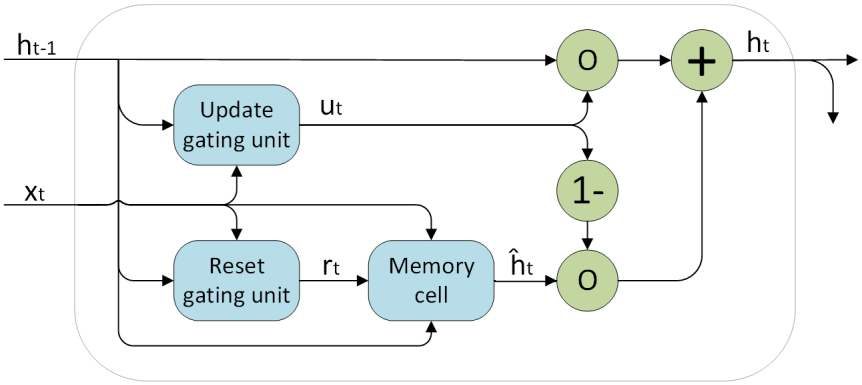

Figure 2.4: High-level diagram of a Gated Recurrent Unit. ◯ denotes element-wise multiplication and *1-* refers to the subtraction of 1 by u_t.

W and **U** are the weight matrices and the (^{c}) denotes it belongs to the memory cell. The size of matrix **W** is *"number of hidden GRU layers"* × *"number of inputs"*, while the size of matrix **U** is *"number of hidden GRU layers"* × *"number of hidden GRU layers"*. $\mathbf{x_t}$ and $\mathbf{h_{t-1}}$ are the respective vectors for the input and the previously hidden layer. ⊙ denotes element-wise multiplication.

The update gating unit is used to decide how much information of *h(t-1)* will carry over to the new value of *h(t)*. A sigmoid activation function is used in generating the output value of unit, that is between zero and one. For a value equal to one, all of *h(t-1)* will carry over to *h(t)* and the candidate for the new *h(t)* will not be used, due to the subtraction by one in the output stage. A gate value of zero, means no information from *h(t-1)* will be carried over to *h(t)* and instead the output of the memory cell is used as the new output value of the GRU. In that way it functions like a gate regulating the flow of *h(t-1)* through the unit.

The update gating unit is defined as [12]

$$u_t = \sigma\Big(\big[\mathbf{W^u x_t}\big] + \big[\mathbf{U^u h_{t-1}}\big] + \mathbf{bias^u}\Big). \tag{2.8}$$

The reset gating unit determines how relevant *h(t-1)* is for the computation of $\hat{h}(t)$ in the memory cell. A sigmoid activation function is utilized in the unit. An output value of one indicates a high relevance while a value of zero means that *h(t-1)* is currently not relevant and by that it will not be used for the computation in the memory cell. In that case, only the current input *x(t)* is impacting the computation in the memory cell, and the reset

gating unit is said to be "resetting" the memory cell for previous stored values or dependencies. The reset gating unit is similar to the update gating unit, though the weights are different. It is defined as in [12]

$$r_t = \sigma\Big(\big[\mathbf{W^r x_t}\big] + \big[\mathbf{U^r h_{t-1}}\big] + \mathbf{bias^r}\Big). \tag{2.9}$$

The output stage is defined by [12]

$$h_t = \big(1 - u_t\big)\hat{h}_t + \big(u_t h_{t-1}\big). \tag{2.10}$$

A GRU layer will usually consists of many recurrent layers. In that case $\mathbf{W}$ and $\mathbf{U}$ are the respective weight matrices in the equations (2.7) to (2.9) while x and h are vectors. As each hidden layer has separate reset and update gating units, each hidden layer will learn to capture dependencies over different time scales. Those units that learn to capture short-term dependencies tends to have a reset gating unit that is frequently active, while those that capture longer-term dependencies will have an update gating unit that is mostly active [12].

In the next chapter, we will discuss the aspects of speech recognition systems, that will have an impact on the system design.

Chapter 3

System Design Aspects

In this chapter, speech recognition systems are discussed in order to understand its impact on establishing the hardware requirements. In the first section, speech recognition systems are explained in more details. Then in section two, the hardware requirements are defined. In the last section, the RNN model that is used as a reference model is presented.

1 Speech Recognition Systems

A computer's ability to identify and respond to sounds or words in the human language is referred to as speech recognition. The applications of speech recognition are many, from trigger word detection and interactive voice response systems to machine translation[1] and transcription. Within the area of speech recognition are acoustic and language modelling. Acoustic modelling is applied to an audio signal in order to create a probability distribution over which phoneme is most likely to represent the given sound. Phonemes are the sounds used to pronounce the words in the human language. *Language modeling is the task of assigning a probability to sentences in a language. [...] Besides assigning a probability to each sequence of words, the language models also assigns a probability for the likelihood of a given word (or a sequence of words) to follow a sequence of words* [15].

The fast paced technological developments within speech recognition,

[1] Machine translation is the field of which a computer is used to translate text or speech from one language to another.

which involves deep learning, have surpassed previous popular methods such as Hidden Markov Models[2] (HMM). Today, recurrent neural networks that is based on either the LSTM or the GRU, is the preferred method for speech recognition systems.

In acoustic modeling the raw input data (audio signal) is first sampled and then decomposed into its respective frequency components. A method for this, is to represent the input data by the Mel-Frequency Cepstral Coefficients (MFCCs). The MFCCs are computed from an audio signal where the complete collection of coefficients, represents a part of the audio signal. Several MFCC methods have been developed through the years with varying sample rates and number of output coefficients. Most commonly is to use a sampling rate of either 8,000 or 16,000 samples per second is used. From the sampled data, frames are created that contains, typically between 20 - 40 ms of data, with an overlap of 10 ms. This means that for a sampling rate of 16,000 samples per second and a frame length of 20 ms, each frame is generated from 320 samples. The MFCC output coefficients are then computed from each frame, resulting in 20 - 40 coefficients, depending on the MFCC method [17]. Another approach is to use a convolutional neural network (CNN) for generating the input to the acoustic model. Here the audio data is sampled at a rate of 44,100 samples per second and frames containing 7.3 ms of data are created. This will typically result in larger input vectors for the RNN model [18][19], up to 256 inputs per vector. The advantages and drawbacks of the different approaches are not considered to be within the scope of this thesis.

In language modeling, the written words are converted into a numerical representation, that can be processed by the RNN. The numerical representation is also referred to as word embeddings. In the area of language modeling, a well-known method for creating word embeddings is the word2vec[3] toolkit, developed by a team of researchers at Google in 2013. The word embeddings created by word2vec is a one-dimentional vector containing features related to the word. The length of the vector can range from 20 and up to several hundreds [20].

For this work to be feasible within the time frame of the thesis's, the focus will be on acoustic modeling, thus language models will not used in the verification of the hardware design. Although, the hardware will be able to process language modeling that utilize one-dimentional word embedding vectors.

[2]A detailed explanation of Hidden Markov Models can be found in [16].
[3]https://code.google.com/archive/p/word2vec/

2 Hardware requirements

The hardware requirements are derived from the previous mentioned aspects related to speech recognition systems. The requirements are detailed below and include the architecture of the RNN, the need for programmable and scalable hardware and the required processing speed.

Architecture: For inference of a complete RNN, the hardware is required to do process the GRU and an FC layer. The GRU is the hidden layer while the FC layer is commonly used as an output layer.

Programmability and scalability: To accommodate the inference of RNN models utilizing different input forms (MFCC and CNN) on the same system on a chip (SoC) design, the hardware is required to process RNNs with various number of inputs, number of hidden layers and number of GRU layers. Also, as the research for a good balance between model accuracy and complexity is ongoing, a hardware implementation should be scalable and reconfigurable [5]. With an increasing number of inputs and layers, the need for memory storage also increases, thus a maximum is set to 256 inputs, 128 hidden layers and 2 GRU layers. The maximum sized RNN then contains a total of 247,681 parameters and when represented in floating point, requires 991 kilo bytes of memory storage.

Processing speed: For real-time processing, the hardware is required to do one classification every 20 ms for MFCC represented inputs and within 7.3 ms for other types of input representations.

3 Reference Recurrent Neural Network

A reference RNN model is needed for verification of the hardware architecture. As the focus of this thesis is not to develop and train a DNN model, a pre-trained GRU-based RNN[4] is used. The application of the model is trigger word detection. This model as illustrated in Figure 3.1, takes 256 inputs per time step, includes two GRU layers, batch normalization and a fully connected (FC) layer. Batch normalization is usually added to a model after a CNN, RNN or a FC layer to normalize the output value. The effect of this, can be an significant increase in learning rate [21]. Training of the model is carried out in Keras, which is a high-level application programming

[4]The model is a part of the Sequence Learning module found on the online learning platform Coursera (https://www.coursera.org).

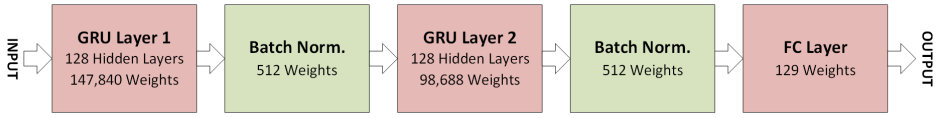

Figure 3.1: Reference model.

interface (API) running on top of TensorFlow. A dataset consisting of 4,000 examples is used for training the model. For evaluation of the inference model, a dataset containing 25 examples is available. Each example has a size of 1,375×256 (time steps × inputs). Evaluating the model on the 25 available examples results in an accuracy of 95%. There are in total 247,681 parameters, all represented in 32-bit floating point.

3.1 Trigger Word Detection

Trigger word detection systems are systems that detects pre-defined keywords, typically from a user, which then "wakes up" or activates other functions or hardware units. Examples of this includes the Amazon Echo with the wake up word "Alexa", Apple Siri can be woken up by "Hey Siri" and Google Home having the keyword "Okay Google". However, these systems does not work without an Internet connection. An RNN speech recognition system can be trained to identify any keyword. An output of such system is given in Figure 3.2. It is a binary output, as any word spoken is either the correct wake up word or not. Since trigger word systems are always on, low power consumption is of great importance. Other models can learn to detect more than one keywords, however this thesis's focus is one trigger word.

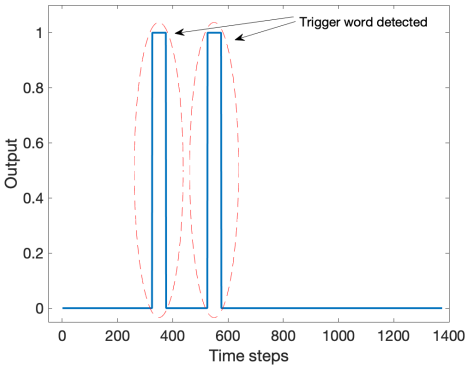

Figure 3.2: Output of the trigger word detection systems detecting one keyword.

Chapter 4

Optimization Techniques

The goal of this thesis is to design a DSP that is integrated with an accelerator, for real-time inference of GRU based RNNs, which has low power consumption and is memory efficient (minimizing the memory storage needed for the application).

Power and energy consumption are important factors in hardware design. Power consumption has an impact on e.g. the size of the power supply and the dimensioning of the interconnecting wires within the silicon chip. Energy consumption is the power consumed over a specified time interval. It has an impact on e.g. the lifetime of a battery. Thus, it is important to minimize the power and energy consumption, when designing hardware for hand-held devices. A more in-depth explanation of power consumption and methods for reducing it, is detailed in the first section.

Reducing the required memory capacity is desirable for reasons such as smaller area in silicon and also for a reduced the power consumption. A memory efficient design is obtained by reducing the word length, that is the number of bits used in representing the data. For this, converting the data type from floating point to fixed point number representation and word length quantization are exploited. This process is detailed in section two and three.

Activation functions are known to be computationally heavy functions, and a reduction in execution time can be obtained by doing approximations of these. This is detailed further in section four.

1 Design for Low Power Consumption

When designing hardware for low power, it is important to look at the components that make up the total power consumption. The total power consumption can be express as the sum of the dynamic, static and direct path power consumption and is given in by [22]

$$P_{total} = P_{dyn} + P_{stat} + P_{dp}\,. \tag{4.1}$$

The static power consumption is caused by a leakage current within the transistor (CMOS technology), when there is no switching activity. It is given by $P_{stat} = I_{stat}V_{dd}$ [22]. Besides the supply voltage, the leakage current is impacted by the threshold voltage of the device. Using a process technology with higher threshold voltage, results in a lower leakage current [22]. However, a higher threshold voltage reduces the performance of the transistor i.e. how fast it can switch from zero to one, which is why considerations should be taken when designing hardware.

The contribution to the direct path power consumption comes from the direct path that occurs between the voltage supply and ground during switching of the transistors (NMOS and PMOS). It will not be addressed in this thesis.

The dynamic power consumption comes from the switching activity when the system is in operation and is given by [23]

$$P_{dyn} = \alpha\, C_L\, {V_{dd}}^2\, f, \tag{4.2}$$

where α is the switching activity, $\mathrm{C_L}$ is the load capacitance, $\mathrm{V_{DD}}$ is the supply voltage and f is the system clock frequency. It is noted that reducing the supply voltage has a quadratic effect on the power consumption. However, there is usually a lower limit for the supply voltage which is set by external factors [22]. Another approach is to exploit parallelism in the design and by that reduce the clock frequency. In [23] it is stated that parallel execution is effective for improving the energy efficiency, as the operations are processed in less time, allowing for a lower clock frequency. By executing a number of operations in parallel, the dynamic power consumption can be reduced by a factor β as given by [23]

$$P^p_{dyn} = \frac{P_{dyn}}{\beta}, \tag{4.3}$$

where β is the number of operation to execute in parallel and P^p_{dyn} is the dynamic power consumption after parallelism. The trade-off for this approach is an increased area in silicon.

The energy dissipation is the power consumed over a specified time period, as given in [23]

$$E(t) = \int_{t1}^{t2} P(t)\, dt. \tag{4.4}$$

Lowering the power consumption by parallel execution will also have a positive impact on the energy dissipation.

Parallelism can be efficiently exploited by moving large amounts of sequential operations from the DSP into an accelerator, that is optimized for parallel executions. In the next section it is discussed how these critical parts of the design are identified.

1.1 Identifying Critical Parts by Model Profiling

When designing an efficient accelerator it is critical to know which parts of the design, that would benefit the most from applying optimization. These parts which are potential bottlenecks in the design, could be computational heavy parts of an algorithm and many sequential operations. To identify the potential bottlenecks of the GRU algorithm and the RNN model, a C-code is developed for execution in a state-of-the art DSP. For this thesis, Tensilica by Cadence Design Systems is made available. Tensilica is a tool used to design and customize processors. The developed C-code is executed on an Xtensa Fusion F1[1] processor. The Fusion F1 DSP is the industry leader for voice-trigger/ wake-on-voice applications [24]. The profiling results are presented in Table 4.1.

Table 4.1: Result of profiling the full sized RNN model on the Xtensa Fusion F1 DSP.

Function name	**Clock cycles**	**Time spent(%)**
GRU	752,684	88.01
∗ Gating unit U and R	252,125	(33.49)
∗ Memory cell	197,672	(26.28)
∗ tanh	298,788	(39.69)
∗ Output stage	4,099	(0.54)
Batch Normalization	99,366	11.61
Fully Connected	527	0.06
Sigmoid function	2,703	0.32

[1]The Fusion F1 is an Xtensa LX7 processor optimized for low power and includes e.g. execution units and instruction sets for floating point operations.

Firstly, it is observed that the most time is spent on processing the GRU unit, which accounts for 88% of the total processing time. Within the GRU, the hyperbolic tangent function takes up most of the processing time followed by the two gating units (U and R) and then the memory cell. Also it is noted, that for the Fusion F1 to do real-time processing of a full sized RNN model, it would need to operate at a rate of 200 million cycles per second.

In order to reduce the processing time, it is decided to design the hardware accelerator for execution of the memory cell, which includes the hyperbolic tangent function and the two gating units, U and R. In this way data level parallelism can be exploited in order to reduce the system frequency and the power consumption. The trade-off for introducing parallel execution is extra hardware, which increases the area in silicon. The batch normalization, fully connected layer, sigmoid function and the output stage of the GRU will be executed in Xtensa processor, that is optimized by adding instructions tailored for the respective functions.

In the next section it is discussed how changing the number representation, is an efficient method for not only reducing the model size, but also reducing the power consumption.

2 Floating point to fixed point conversion

Converting the data type from floating-point to fixed-point representation is an efficient methods for reducing the word length. Significant improvements may be achieved in terms of reduced area and power consumption and higher performance. For example, a reduction of the cycle count by 75% and of the energy consumption by 76% has been reported for an MPEG-2 video compression algorithm [23]. The conversion is a trade-off between model accuracy[2] and implementation costs like area and power consumption. It is therefore necessary to analyze the loss in accuracy versus the parameters used for the fixed point conversion, that is the word length and the position of fractional point [23].

However, for binary classification and with data sets having a significant disparity between positive and negative labels, the accuracy can be a misleading metric. Take e.g. the output of the FC layer for a single example of the data set, as presented in Figure 4.1. Here the accuracy for the fixed point model is calculated to 95%. But in the case where the probability is zero for all 1,375 time steps, the accuracy would still be above 90% while two

[2]See Appendix A for definition and more details about accuracy

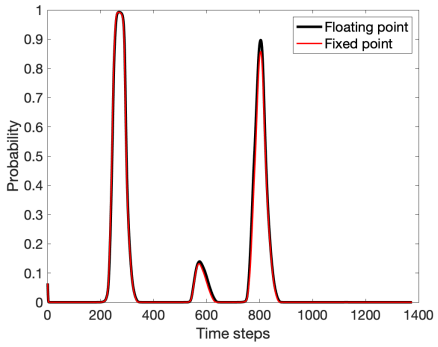

Figure 4.1: Output of fully connected layer followed by a sigmoid function.

positive predictions/trigger words are missed. As a result, when converting into fixed-point representation, the following parameters for a floating-point and a fixed-point model will be compared.

- TP: True Positives - (model predicted correct, positive '1')
- TN: True Negatives[3] - (model predicted correct, negative '0')
- FP: False Positives - (model predicted positive, but reality is negative)
- FN: False Negatives - (model predicted negative, but reality is positive)

Two models are created in MATLAB, a reference model using floating-point numbers and a fixed-point model. In fixed-point the numbers are represented in two's complement using 16-bit (2 integer bits and 13 fractional bits). The comparison of the two models is presented in Table 4.2. As

Table 4.2: Model predictions.

	Floating point	**Fixed point**
TP	20	20
TN	N/A	N/A
FP	1	1
FN	27	27

floating-point numbers are represented with 32-bit, the model size can be reduced with 50% by converting into 16-bit fixed point numbers. This is

[3]The number of true negatives is not available from the validation dataset.

while keeping a prediction rate that is the same as for floating-point data types. Another method to further reduce the word length is quantization. The use of this is discussed in the next section.

3 Quantization

Recurrent neural networks operates with a large amount of parameters. The reference model used for this thesis includes two GRU layers, which in total accounts for 246,528 parameters. It would require 987 kilobytes of memory to stored every parameter as floating-point and 494 kilobytes for 16-bit fixed-point numbers. Being able to reduce the number of bits necessary for each parameter, without affecting the number of correct detections, would have a positive impact on the area and power consumption. Quantization is a method for reducing the word length. According to previous studies [25] [26], quantization can be applied to the RNN parameters in order to reduce the word length down to 8-bit. This is without causing a considerable decrease in the accuracy and without retraining the model or using other techniques such as quantization aware training [26]. On the other hand, applying quantization for reducing the word length to e.g. 4-bit would require retraining of the model or the use of quantization-aware training which is not under the scope of this thesis.

Quantization can be explained as mapping a large number of inputs into a smaller set of values. In Figure 4.2, a floating-point number is first converted into a 4-bit fixed-point number and then it is quantized using 2 bits. Some of the original information is lost in the process, which is referred to as the quantization error and it should be taken into consideration when applying quantization.

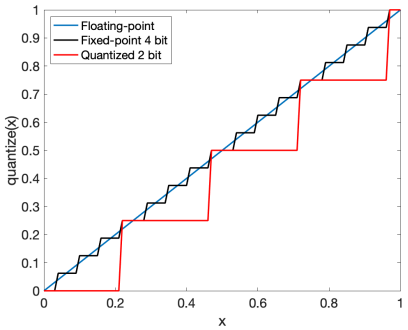

Figure 4.2: Quantization of a 4-bit fixed-point number into 2-bit word.

All of the 246,528 GRU parameters currently represented by 16-bit fixed-point numbers are quantized into 8-bit and the prediction rate obtained from the MATLAB model is equal to those presented in table 4.2. By applying quantization the required memory storage has been further decreased to 246 kilobytes resulting in a total reduction of 75%.

In the next section, the approximations of activation function will be detailed.

4 Activation Functions

Activation functions are computationally heavy functions. The sigmoid and the hyperbolic tangent, as defined in equation (2.2) and (2.3), requires division and exponentiation, that are not practical for implementing in hardware [27]. In addition, as it was observed during profiling of the network, the hyperbolic tangent function occupies the largest amount of the processing time. In this section it is discussed how the hyperbolic tangent and the sigmoid functions are approximated for efficient hardware implementation.

4.1 Hyperbolic Tangent Approximation

The hyperbolic tangent is the activation function used in the memory cell of the GRU layer. Common approaches used for hardware implementation includes piecewise linear approximation and look-up tables (LUTs) [27]. LUTs are faster but take up more area in silicon while piecewise linear approximations are slower but they utilizes a smaller in area [27]. An optimized approach based on the LUT, is the range addressable lookup table or RALUT. It is faster than a LUT and consumes less area [28]. In a LUT, every stored table value corresponds to a single input. For a RALUT, every stored table value corresponds to a range of inputs. This approach allows for a great reduction in the number of required table values, especially if the table values are similar over a range of inputs, which is the case for the hyperbolic tangent function [28].

In this thesis, the hyperbolic tangent function is approximated by using the RALUT approach. A fixed-point model easily adaptable for hardware implementation was created in MATLAB. By exploiting the symmetrical behavior of the function, the approximated output is determined based on

the input divided into three segments, given by:

$$tanh_{approx}(x) = \begin{cases} x & : \quad 0 \leq x < 0.26 \quad (Seg.1) \\ x - ralut_{out}(x) & : 0.26 \leq x < 2.30 \quad (Seg.2) \\ 1 & : \quad x \geq 2.30 \quad (Seg.3) \end{cases}$$

For the second segment, the table values stored in the RALUT, are calculated as:

$$ralut_{out}(x) = \begin{cases} x - tanh(x) & : \quad 0.26 \leq x < 1.00 \\ 1 - tanh(x) & : \quad 1.00 \leq x < 2.30 \end{cases}$$

By using 92 table values, each represented by 11 bits, a maximum error of 0.006 is obtained for segment one and two and 0.02 for the third segment. In Figure 4.3 is the error presented for an input vector containing the numbers from 0 to 3 with a step size of 0.001. A total of 1,012 bits are required for the RALUT. According to [27] this approach can provide an output in 2.12 ns, meaning it can complete in one cycle when the operation frequency is less than 470 MHz. A great performance improvement is gained but the trade-off is an increased area in silicon, required for the logic used to store the 92 table values.

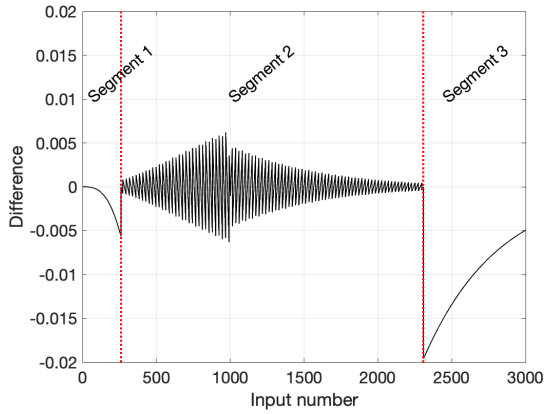

Figure 4.3: Difference between MATLAB tanh and RALUT approximation.

4.2 Sigmoid Approximation

A sigmoid function is utilized as the activation function in the GRU and for the two gating units *update* and *reset*, as detailed in chapter 2. For

the reference model that was developed in Keras, the sigmoid function is approximated using a hard sigmoid, given by:

$$hardsigmoid(x) = \begin{cases} 0 & : \quad x < -2.5 \\ 1 & : \quad x > 2.5 \\ 0.2 \times x + 0.5 & : -2.5 \leq x \leq 2.5 \end{cases}$$

A hard sigmoid is more suitable for hardware implementation as it does not include division and exponentiation. As for the reference model, a hard sigmoid will be utilized for the two gating units in the hardware accelerator.

The FC layer also includes a sigmoid activation function. As the FC layer will be executed in the DSP, the sigmoid function is implemented by a piece-wise polynomial approximation as proposed in [29]. This approach is simple to implement in the processor and it is faster than the sigmoid function. Also, it is better suited for a fixed-point representation that is based on integer data types. The piece-wise polynomial approximation is designed to have a maximum error of 0.021, as illustrated in Figure 4.4 and it is defined as [29]:

$$poly_{approx}(x) = \begin{cases} 0 & : \quad x \leq -4 \\ 0.5 \times (1 - |0.25 \times x|)^2 & : -4 < x < 0 \\ 1 - 0.5 \times (1 - 0.25 \times x)^2 & : \quad 0 \leq x < 4 \\ 1 & : \quad x \geq 4 \end{cases}$$

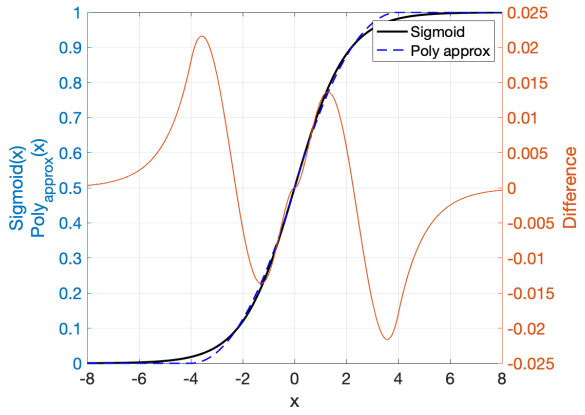

Figure 4.4: Comparison of the sigmoid function and the piece-wise polynomial approximation.

In the next chapter follows a detailed description of the design and configuration of the DSP.

Chapter 5

DSP Design and Implementation

To design a DSP, the software tool Tensilica by Cadence is made available for this thesis. From this tool, the Tensilica processor Xtensa LX7 together with the data book is used as a framework for designing an optimized DSP. The Xtensa LX7 is a 32-bit RISC processor based on the Harvard architecture[1]. It is configurable and extensible[2] allowing processor designers to tailor the architecture to specific applications. Changing the processor's instruction set, memories, and interfaces can make a significant difference in its efficiency and performance, particularly for the data-intensive applications that represent the "heavy-lifting" of many embedded systems [31].

A high level diagram of the DSP and the hardware accelerator design is presented in Figure 5.1. As illustrated the main functionality for the DSP, is the execution of the output stage of the GRU, the batch normalization layer as well as the FC layer including the sigmoid approximation. For a low power design it is important to keep the number of cycles needed in processing these functions at a minimum. It is also required that the DSP can process data that is represented in fixed-point notation. For this to be feasible, all unnecessary units in the Xtensa LX7 configuration is taken out, while application specific instructions, special registers, interface ports and software are designed. By adding tailored instructions to the instruction set architecture of the DSP, the processor executes the target application code

[1]This term is used today to apply to machines with a single main memory but with separate instruction and data caches [30].

[2]Extensibility's goal is to allow features to be added or adapted in any form that optimizes the processor's cost, power consumption, and application performance [31]

in fewer cycles. This allows for running the DSP at a lower clock frequency and possibly also at a lower supply voltage, which in turn reduces the power dissipation and the energy consumption.

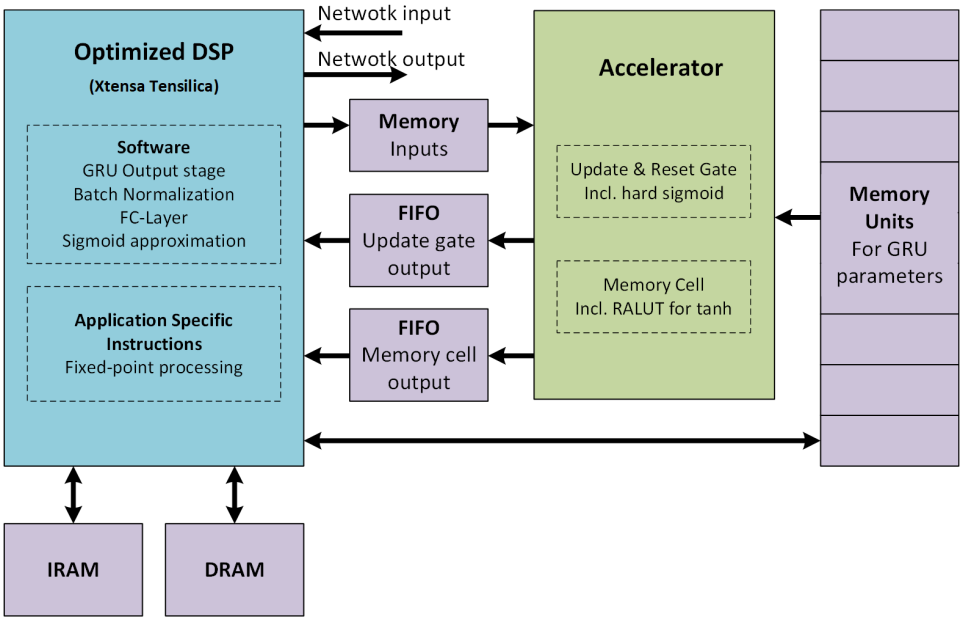

Figure 5.1: High level block diagram of the system architecture.

First-In First-Out (FIFO) storage units are utilized for transferring data from the hardware accelerator to the DSP. Only a small FIFO is necessary as just one 16-bit word is transferred at a time. When a FIFO read is initiated from the software, the processor will stall the pipeline and enter a low power mode until a ready signal is given from the FIFO, indicating that new data is available. If the processor is required to work on other tasks while waiting for the accelerator, an interrupt would have to be configured. However this would also increase the power consumption.

A memory unit is used to provide data from the DSP to the accelerator. This allows for the DSP to upload the next input data, while the accelerator is processing data. To avoid unnecessary data flow between these two units, a direct connection is made between the accelerator and the memory units containing the network parameters.

The changes made to the configuration, creation of application specific instructions and the software are discussed in the following sections.

1 DSP Configuration

The modifications made to the configuration has led to the DSP architecture as illustrated in Figure 5.2. The physical area is 0.106 mm^2 including 2 kilobytes of instruction RAM and 8 kilobytes of data RAM. The estimated power consumption is 184 μW for an operation frequency at 1.5 MHz. The operation frequency can be scaled up to 430 MHz, with an estimated power consumption at 18.75 mW.

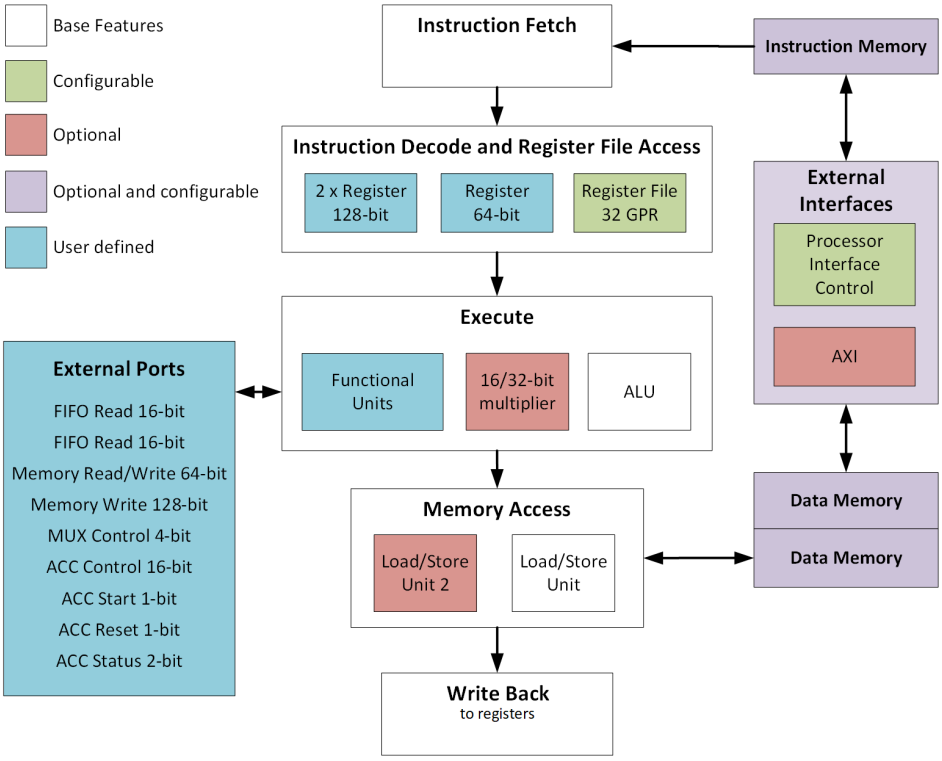

Figure 5.2: Block diagram of DSP architecture.

Local storage units for the data and the instructions can be configured in several different ways using ROM, RAM and/or caches. According to [31], using instruction RAM and data RAM is more power efficient than using instruction and data caches of a similar size, as there are no tag arrays and no tag comparisons to determine a cache hit or miss. For this design one instruction memory (SRAM) and two data memory units (SRAM) are configured.

The pipeline length is configurable for either five or seven stages. A

length of five is configured to match the access latency for the memory units used in this design. A seven stage pipeline, includes an extra memory stage which provides an extra clock cycle for slower memory units.

The register file is configured with 32 general purpose registers (GPR). There is an option to extend the size to 64 registers. Some applications will executed faster[3] with more registers. The drawback is that more gates are used to implement the extra registers. Based on the results from the software profiling, it was not found necessary to add more GPRs. Three application specific registers at respectively 1x64-bit and 2x128-bit are designed as temporary storage for data that are to be written to the memory units containing the GRU parameters (64-bit) and for the memory unit storing the inputs for the accelerator (128-bit).

A secondary load/store unit is added to the configuration to match the two data memories and it allows for two memory operation at the same time.

External ports are created for interfacing the processor to the FIFO's, the hardware accelerator (control and status signals) and for the input and output data of the RNN.

An AXI-bus is added for access to off-chip memory. Assuming new network parameters can be downloaded in this way, although this functionality is not within the scope of this thesis.

New functional units are automatically inferred into the pipeline by the software, when creating application specific instructions that involves computations. In the next section are the application specific instructions detailed.

1.1 Application Specific Instructions

Application specific instructions are designed in order to reduce the number of cycles needed for the computations in the the output stage of the GRU algorithm, the batch normalization, the fully connected layer and for processing 16-bit fixed point computations.

Three instructions are designed to perform the computations that are executed in the output layer of the GRU algorithm. The algorithm was detailed in chapter 2, equation (2.10) and repeated here for simplicity:

$$h_t = \big(1 - u_t\big)\hat{h}_t + \big(u_t\, h_{t-1}\big).$$

[3]If the number of registers required exceeds the number of physical register, excess registers will be moved to memory, which is more time consuming.

One instruction is performing the element-wise multiplication of the two vectors, u_t and h_{t-1}. They are both 1-dimensional vectors with *"the number of hidden GRU layers"* elements. The second instruction is performing the subtraction, $1 - u_t$. While the last instruction is doing the remaining multiplication and addition.

The batch normalization was introduced together with the reference model in chapter 3, section 3. It is given by

$$output = \frac{(input - mean)}{\sqrt{variance + \epsilon}} \times \gamma \ + \ \beta \quad , \quad \epsilon = 0.001 \tag{5.1}$$

where *mean*, *variance*, *gamma* and *beta* are parameters obtained from training the model.

Three mathematical operations including square root and division can be avoided, by reducing the expression into

$$output = (input - mean) \times factor + \beta \tag{5.2}$$

and calculate the value for *factor* off-line. This will have a significant impact on the speed up of the batch normalization layer. One instruction is created to perform the batch normalization as given by (5.2).

For the FC layer, which is given by

$$output = \sum_{t=1}^{n} input(t) \times weight(t) + output(t-1), \tag{5.3}$$

an instruction is created to do the multiply and accumulate operation. The fixed-point representation has been extended with two integer bits to avoid overflow.

Finally, an instruction is created to perform the piece-wise polynomial approximation of the sigmoid function as described in chapter 4, section 4.2. This instruction generates an 16-bit output, to keep the word length of the network output, the same as the initial network input.

1.2 Software Implementation

The functionality of the software developed for the DSP, is illustrated in Listing 5.1. The code makes use of the application specific instructions described in the previous section.

```
int main()
  for (counter <= number_of_GRU_layers)
    while (counter < number_of_hidden_layers)
        read new value from FIFO gate u
        calculate Gate_u * h(t-1)
        calculate 1 - Gate_u
        counter + 1

    while (counter < number_of_hidden_layers)
        read new value from FIFO memory cell
        counter + 1

    while (counter < number_of_hidden_layers)
        h(t) = multiply and add
        bn_out = batch normalization
        if (1 layer GRU architecture)
            update register_ht with h(t)
            perform fully connected calculation
            when (counter modulo 8 == 0)
                write register_ht to memory
        else  // 2 layer GRU architecture
            if (GRU layer 1)
                update register_xt with bn_out
                update register_ht with h(t)_layer_2
                when (counter modulo 8 == 0)
                    write register_ht to memory
                    write register_xt to memory
            else  // GRU layer 2
                update register_ht with h(t)_layer_1
                perform fully connected calculation
                when (counter modulo 8 == 0)
                    write register_ht to memory
        counter + 1
  Sigmoid function
  return 0;
```

Listing 5.1: Pseudo code developed for the DSP software.

1.3 Profiling

The C-code running on the optimized processor was profiled to obtain the number of clock cycles required for each function. The result is presented in Table 5.1.

Table 5.1: Result of profiling DSP optimized for the GRU.

Function name	**Fusion F1**	**Optimized**	**Speed Up**
GRU Output stage	4,099	1,293	x3,17
Batch Normalization	99,366	3,712	x26.76
Fully Connected	526	896	x0.41
Sigmoid function	2,703	35	x77.22
Total	106,694	5,936	x17.97

In general improvements are achieved for the optimized processor and as expected, with the largest gain achieved for the optimized batch normalization and the approximated sigmoid. An interesting result is that no speed up was achieve for the FC layer. However, the higher amount of cycles (370) used for the FC layer, has little impact on the total speed up achieved. The time spent on improving the C-code for the FC layer, should be considered with respect to how much extra speed up can be gained.

In the next chapter, the design and implementation of the hardware accelerator is explained.

Chapter 6

Accelerator Design and Implementation

This section focuses on illustrating the implementation of the GRU layer as a generic hardware accelerator. First, the level of applying parallelism is discussed. Then the general architecture of the design is given including the signals received from outside. Secondly, main modules are explained in detail. Lastly, the memory organization is given in order to complete the general understanding of the hardware accelerator as a whole and provide a better grasp of how many memory units are used and where.

1 Level of parallelism in Accelerator

The purpose of the accelerator is to reduce the system operation rate[1] in order to obtain a lower power consumption. The idea is to execute arithmetic operations in parallel instead of having sequential operations at a higher speed. From the profiling results, it was found that the two gating units and the memory cell contains many sequential operations and in addition, the memory cell also includes the computational heavy hyperbolic tangent function. For a maximum sized RNN, there are in total 147,968 sequential arithmetic operations (multiplications and additions), not counting the activation functions. The equations making up these three units, as defined in chapter 2, are listed here:

$$
\begin{aligned}
u_t &= \sigma\Big(\big[\mathbf{W^u}x_t\big] + \big[\mathbf{U^u}h_{t-1}\big] + bias^u\Big),\\
r_t &= \sigma\Big(\big[\mathbf{W^r}x_t\big] + \big[\mathbf{U^r}h_{t-1}\big] + bias^r\Big),\\
\hat{h}_t &= \phi_h\Big(\big[\mathbf{W^c}x_t\big] + \big[\mathbf{U^c}\big(r_t \odot h_{t-1}\big)\big] + bias^c\Big).
\end{aligned}
$$

From the equations, it is observed that the two gating units, U and R, are completely independent and can be executed in parallel, while the memory cell is dependent on the output from the reset gating unit. Although, by starting the computations of $\mathbf{W}$ multiplied by *x(t)*, the execution of the memory cell can be started together with the gating units. In that way, only the second part of the equation remains to be computed, which is $\mathbf{U^c}\big(r_t \odot h_{t-1}\big)$. Also memory access is reduced in this way. The total number of sequential operations are now reduced to: *the number of operations required for the reset gating unit plus the number of operations needed for the second part of the memory cell.* For a maximum sized RNN this is 66,176 operations and 49,792 operations for 128 inputs while for 40 inputs (which is the number of inputs for a complete set of MFCC) it is 38,528 operations.

In column two of Table 6.1, the number of arithmetic operations required by the accelerator are presented, when extending the parallelism to include processing of more than one input at a time. The total number of arithmetic operations required for processing the complete RNN for one time step is found in the third column. For this calculation it is assumed that the LUT used for the hyperbolic tangent approximation, takes one cycle per look up. The hard sigmoid function is assumed to take two cycles per input. The number of hidden layers is assumed to be 128. In the last column is rate of

[1]This is the system operating frequency.

which the system is required to operate at, to finish the computations in 7.3 ms.

Table 6.1: Impact on system operation rate for different levels of parallelism with 256 inputs.

Level of	Arithmetic Operations		System
parallelism	Accelerator	Full RNN	operation rate
1	66,176	127,430	17.5 MHz
4	16,800	41,052	5.7 MHz
8	8,528	26,726	3.7 MHz

The same data for 40 inputs are presented in Table 6.2. It is noted that by continuing to increase the number of parallel executions, the computational bottleneck is moved from the accelerator and into the DSP. For a full sized RNN and processing of eight input in parallel, the total number of operations for the accelerator is 14,940 and 11,786 for the DSP. This provides for balance system with respect to the computational workload. It is decided to design the accelerator for computing eight inputs in parallel and also the two gating units and the memory cell in parallel. The trade-off for introducing parallelism, is an increased area in silicon due to the extra hardware required for the accelerator.

Table 6.2: Impact on system operation rate for different levels of parallelism with 40 inputs.

Level of	Arithmetic Operations		System
parallelism	Accelerator	Full RNN	operation rate
1	38,016	99,270	5 MHz
4	9,600	33,862	1.7 MHz
8	4,864	22,998	1.2 MHz

2 General Architecture

The hardware accelerator is designed to perform parallel execution of the two gates, update and reset, and the memory cell. The input and output ports of the top module of the accelerator and the three main modules included can be seen in Figure 6.1. The design is scalable which allows for changing the number of inputs and hidden layers. This is controlled from the DSP which also provides information about which GRU layer is to be processed and starts the accelerator.

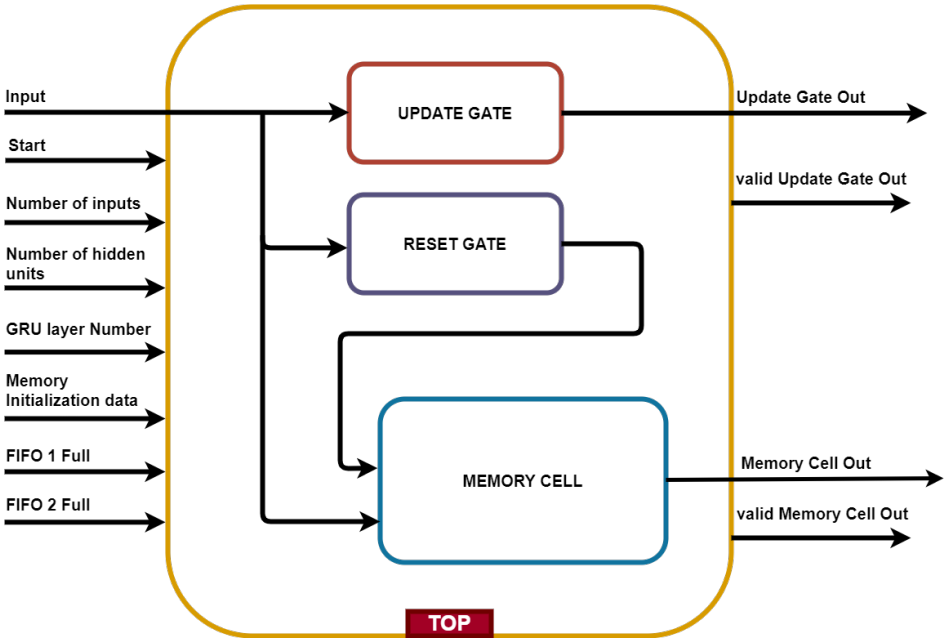

Figure 6.1: General architecture of the hardware accelerator

Within each module are eight inputs computed in parallel and the activation function is also applied, which is illustrated in the more detailed diagram of the hardware accelerator in Figure 6.2. Each module has direct access to memory units, containing the respective weights. Initialization of these memory units is handled by the DSP.

The update gate (update gating unit) and the reset gate (reset gating unit) will be detailed in the following section, since they are very much alike. After the gates are explained, the next section talks about the memory cell and how it is connected to the other modules. The last section in this chapter

explains how the control modules (Input Control Unit(CU) and reset gate output(Rr) Memory Bank) is organized. In addition, the organization of the other memories that are necessary for the design, such as the memories for the weights, are explained.

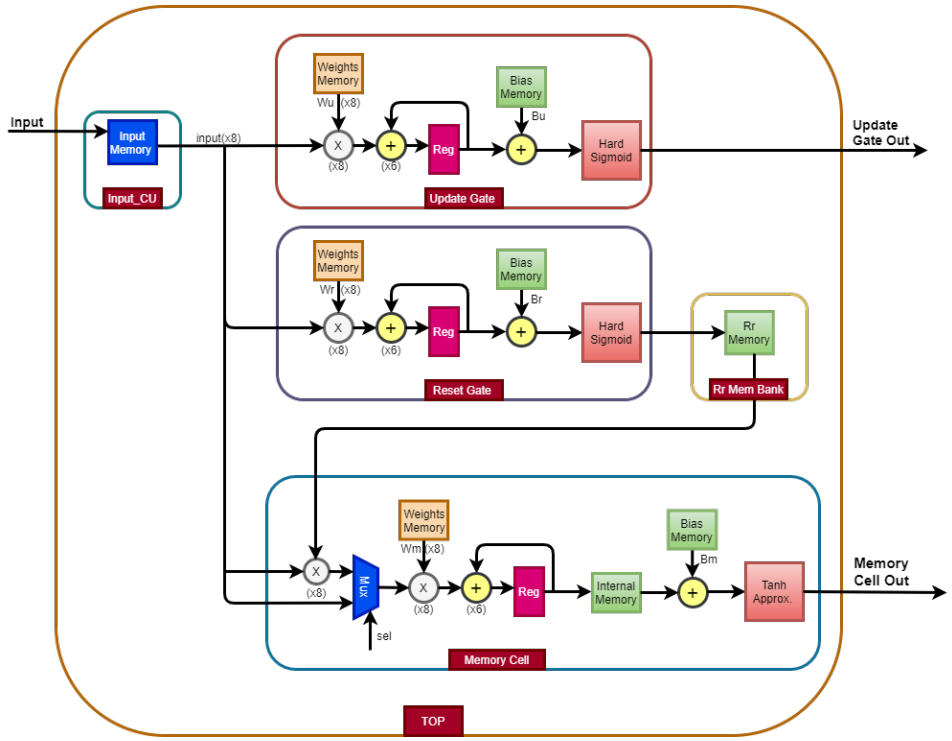

Figure 6.2: Detailed architecture of the hardware accelerator

3 Gate U and R

The update and reset gates are built in a similar way, as the only difference between them is that they read the weights from different memory units. For that reason the architecture and functionality will be detailed step by step, with respect to the reset gate.

After a start signal is given by the DSP, then for every clock cycle, eight inputs and eight weights are read from the respective memory units. As illustrated in Figure 6.3, they are multiplied in parallel and added together. The output of the last adder is stored in a register, to be accumulated with

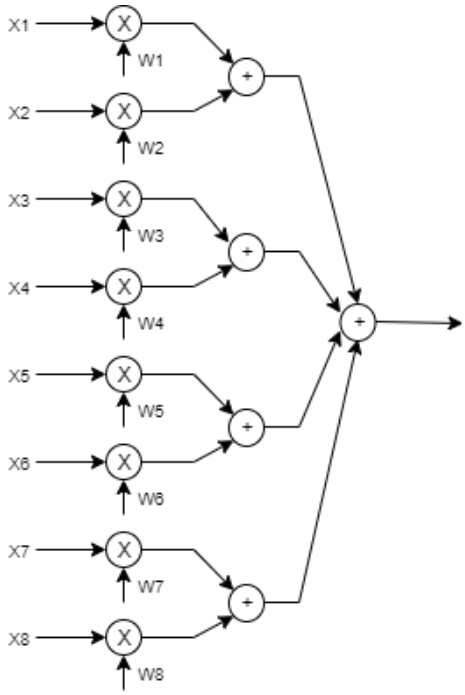

Figure 6.3: Optimized tree architecture for the multiplication and addition

the next value. The number of accumulations depends on the number of inputs and hidden layers. For a full sized model, there are a total of 256 inputs (*x(t)*) + 128 recurrent inputs (*h(t-1)*), resulting in 384 inputs. Because of the eight parallel multiplications, a total of 48 (384/8) values are accumulated. Then the bias is added to obtain a valid input for the sigmoid unit. As stated in Chapter 4, section 4.2, the sigmoid function is approximated as a hard sigmoid to reduce the computational complexity and the processing

time. After the sigmoid unit, the outputs of the reset and update gates are ready and a valid signal is sent to other parts of the design.

There are also registers in between all the multipliers and adders in order to decrease the critical path between units and introduce pipelines.

For the update gate the outputs are stored in a FIFO, from where it is read by the DSP for further processing of the GRU output stage. The outputs of the reset gate are stored in a scratchpad memory as they will be multiplied with *h(t-1)* and used in the memory cell. Since the memory that stores *h(t-1)* values has a bandwidth for 8 values, the outputs of the reset gate are stored in the same manner and read from the memory 8 by 8 for using in the memory cell. The bandwidth of this memory unit is thus 128-bit and it has 16 addresses for storing 128 16-bit values.

4 Memory Cell

In the memory cell, the first part where the *x(t)* inputs are multiplied with the weights, is the same as the other two gates. However, instead of using *h(t-1)* for the rest of the weights, the outputs of the reset gate are multiplied by *h(t-1)* and used. Since the outputs of the reset gate that should be used for this multiplication is not ready instantaneously, the memory cell is kept idle when the other gates are multiplying *h(t-1)* with the corresponding weights. This makes the design of the memory cell different from the other two gates.

The first 256 inputs that are coming from the input memory are still needed for the memory cell. In order to use the inputs that are already read for the update and reset gates, the inputs are given to memory cell at the same time with the other gates and multiplied with the corresponding weights. This benefits the design by decreasing the number of memory reads and decreasing the number of clock cycles necessary to complete the memory cell. After 32 clock cycles (256/8=32), all the *x(t)* inputs are multiplied with corresponding weights, the memory cell is put into idle mode. The reset and update gates computes multiplications of 128 *h(t-1)* inputs and weights in 128/8=16 clock cycles. When the first row is ready (after 32+16=48 clock cycles) and accumulated, the output is written into a memory to be stored. This process where the multipliers are used to multiply inputs with weights and idling the multipliers is done 128 times. At the end of each 48 clock cycles, an incomplete accumulated value is obtained and written into the memory. This is the first part.

When the reset gate is completed fully, all the values are ready in the scratchpad memory and read 8 by 8 for usage in the memory cell. The *h(t-1)* values are also read from the input memory by the help of "Input Control Unit". The outputs of the reset gate and *h(t-1)* values are multiplied 8 by 8. The weights for this second part is read from the weight memories. Since there are 128 values, it takes 16 clock cycles to finish the calculation for the first row. All the values are accumulated and they are added to the stored outputs of the first part and the bias for the corresponding row. When a row is completed, the accumulated output is given to tanh module which is implemented as a RALUT (which is explained in chapter 4, section 4.1). The output of the tanh function is the valid output coming from the memory cell and it is given to the processor with a valid signal. In total of 128 outputs are given.

5 Memory Organization

In general, data movement in deep neural networks designs, dominates the energy consumption and in order to decrease the energy consumption, the key element is to have an efficient memory hierarchy and a well-built dataflow [32]. In order to accomplish this, several techniques are applied as it will be explained in this chapter. There are two memory control units in the design, to handle the flow and timing for reading data from the input memory and the scratchpad memory. Before explaining how these control units work, let's look at the organization of the memory units. A general architecture for all of the memories and how they are connected in the system is given in Figure 6.4.

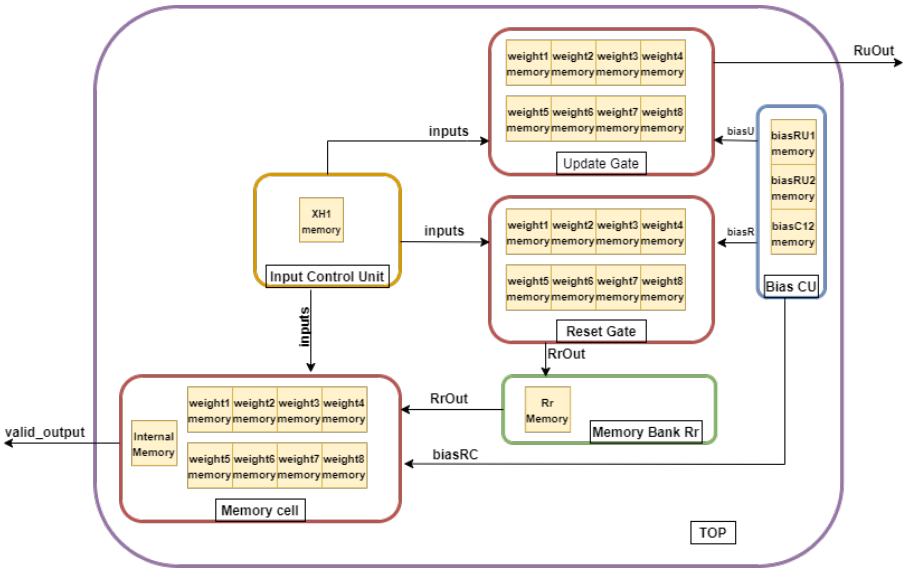

Figure 6.4: General Memory Architecture

For a full sized RNN model, there are 384×128 = 49,152 weights to be stored in both of the update and reset gate and in the memory cell. In order to create an efficient memory architecture, several smaller memory units is used instead of larger units. The memories used have 16-bit storage per address. In each address are stored two weights represented by 8-bit each. To accommodate the parallel execution of eight inputs, four memory units are wrapped together to obtain 64 bit bandwidth, see Figure 6.5. As a result, there are in total 4×3=12 memory units for the first GRU layer and another 12 for the second GRU layer.

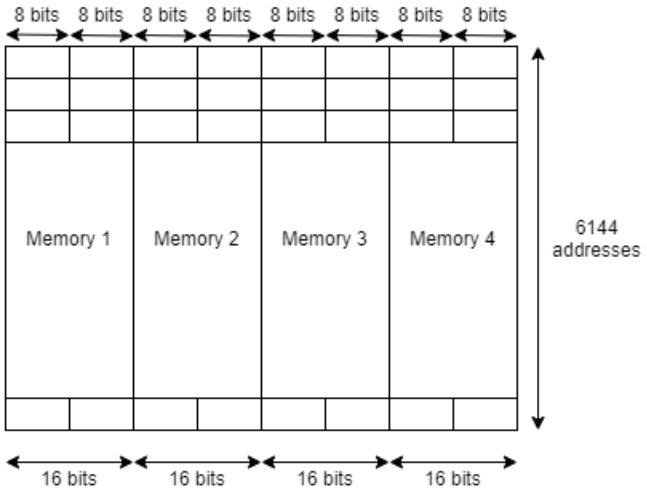

Figure 6.5: Memory allocation for the weights in the design

A similar approach is taken for the memory units used for the accelerator inputs (*x(t)* and *h(t-1)*). In order to read eight inputs that are 16-bit at a time, the memory has a bandwidth of 128-bit and 48 addresses. New data is written from and handled by the DSP. For the biases, three memory units with 16-bit bandwidth and 128 address spaces, are used. Biases for the update and reset gate are stored together and one memory unit is used for each GRU layer. For the memory cell the biases for both GRU layers are stored in one memory.

Input Control Unit: This unit is responsible for reading all the 384-inputs consecutively until it has read *"the number of hidden units in the design"* times. After this is done, it waits for a control signal to be received from the memory unit and it reads the *h(t-1)* inputs again with the same number of times. After these calculations are done, the memory is rewritten by the processor to be ready for the next calculation.

Reset Output Unit: This unit is responsible for storing the valid reset outputs and reading them 8 by 8 when the memory cell gives a signal. This signal is aligned with the one for the Input Control Unit which reads the *h(t-1)* for the second time since they are multiplied before being used as a input to the memory cell.

This concludes the explanation part of designing the hardware accelerator and the next section presents the results of this implementation. Discussions about the results are also included in the next section.

Chapter 7

Implementation Results and Discussions

In this chapter the functional simulation and the outcome is discussed. Then the design is evaluated in terms of area, performance, the power and energy consumption. A layout is presented afterwards. Finally the design is compared to other systems.

1 Functional simulations

Functional simulations are carried out in order to verify, that the functionality of the developed hardware correspond to the fixed-point model in MATLAB. This ensures that the same number of correct predictions are achieved in hardware as with the fixed-point model. Due to complications with the Tensilica tool, no synthesis are performed for the DSP. A consequent of this, is that no simulations are performed for the complete system, as it is illustrated in Figure 5.1. Instead, the functionality of the DSP and the accelerator are simulated independently and the outputs are compared to the fixed-point model. Also, the cycles used for each unit are counted. All simulations are performed on a maximum sized network containing 256 inputs, 128 hidden layers and 2 GRU layers.

1.1 DSP

To verify the functionality of the DSP, the Tensilica tool is used. The functionality to be verified is summarized here,

1. Read data from the two FIFOs, GRU layer 1

2. Compute output stage of the GRU and store result in memory
3. Compute batch normalization
4. Read data from the two FIFOs, GRU layer 2
5. Compute output stage of the GRU and store result in memory
6. Compute batch normalization
7. Compute FC layer and sigmoid

Reading data from the two FIFOs are simulated by reading data from files. The files containing the input data, Γu and $\hat{h}(t)$, are generated in MATLAB using the fixed point model and a validation data with 1375 time steps. After processing of the data, the output *h(t)* is writing to a file, simulating a memory write operation. The outputs of the GRU layers and the batch normalization functions are compared to the fixed point model in MATLAB. A simulation is started at time step one with the outputs of time step two presented in Figure 7.1 to Figure 7.4. The simulation showed a 100% equality for all time steps. For the memory write operation, the address and the

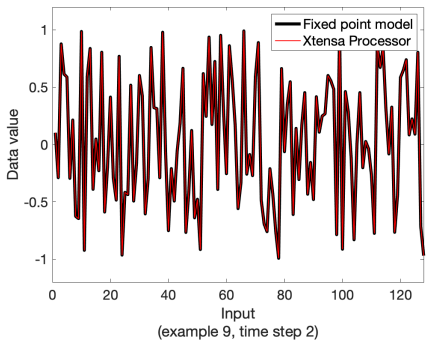

Figure 7.1: GRU Layer 1

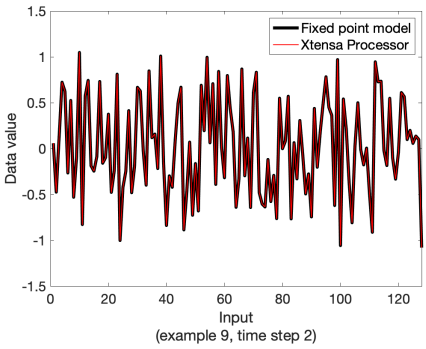

Figure 7.2: Batch normalization 1

corresponding data are written to a file. The address is an 8-bit value and the data, which is *h(t-1)* and for GRU layer 2 also *x(t)*, consists of eight 16-bit values that forms one 128-bit word. The address and data are validated with MATLAB.

The output of the FC-layer for time steps one and two, matches the fixed point model.

	Time step 1	Time step 2
Fixed point model:	-3.188	-3.529
DSP:	-3.181	-3.517

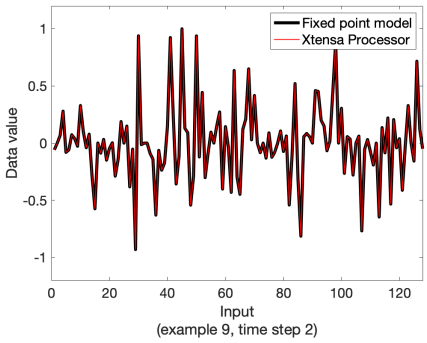

Figure 7.3: GRU Layer 2

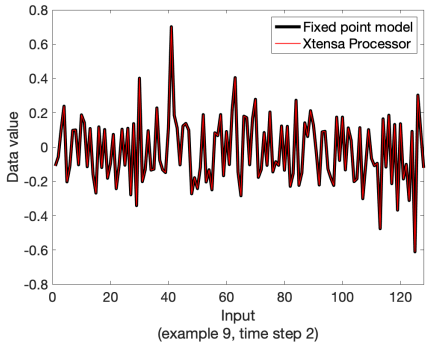

Figure 7.4: Batch normalization 2

The sigmoid approximation implemented in the DSP is tested in the range from -4.2 to 4.2 with a step size of 0.02. The output is compared to a fixed point model created in MATLAB. A maximum difference of 0.12×10^{-3} was found, as illustrated in Figure 7.5.

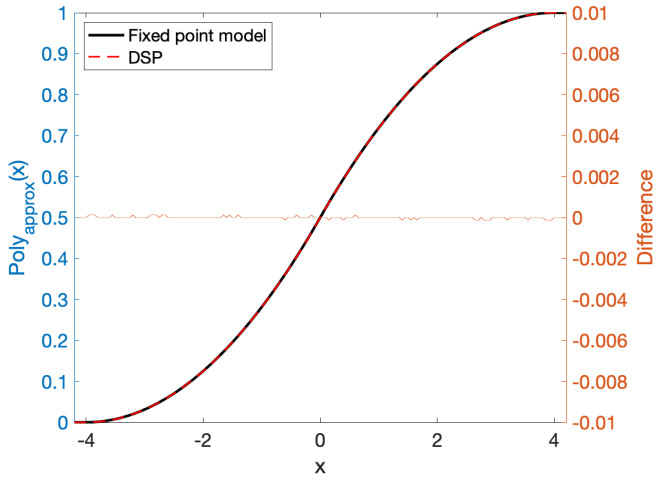

Figure 7.5: Verification of implemented sigmoid approximation in DSP versus MATLAB model.

1.2 Hyperbolic Tangent

The approximation for the hyperbolic tangent function, that is implemented as a RALUT, is verified by an input vector containing the numbers from

-3 to 3 with a step size of 0.004. The output of the RALUT is compared to the MATLAB tanh[1] function. As presented in Figure 7.6, a maximum error of 0.06 was found for segment one and two, while for the third segment the maximum error is 0.02. This is within the maximum specified error, that was defined for the look up table implementation. In Figure 7.7 is the output of the RALUT compared to the MATLAB tanh function. From the simulation, it was found that it takes one clock cycle to retrieve an output from the RALUT.

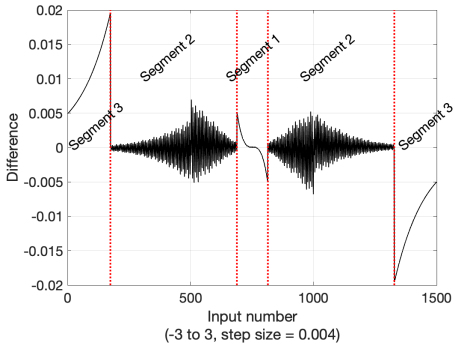

Figure 7.6: Difference between RALUT implementation and MATLAB tanh.

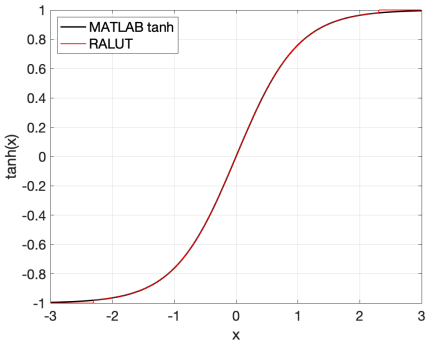

Figure 7.7: RALUT output vs. MATLAB tanh output.

1.3 Hardware accelerator

The verification of the hardware accelerator is done by generating a test bench for the top module and simulating results in Questasim Advanced Simulator by Mentor Graphics.

Table 7.1: Accelerator cycle count.

Function name	Clock cycles
Gate U and R	6,159
Memory cell	8,216
tanh	1

[1]tanh is the built-in hyperbolic tangent function in MATLAB.

2 Area, Power and Performance Evalutaion

In this section the design is evaluated in terms of the area, performance and the power and energy consumption.

2.1 Area

The area was obtained from synthesis of the design using the ST Microelectronics 28nm FD-SOI technology at a frequency of 430 MHz and supply voltage at 0.9V. For real-time processing of speech data, the frequency can be scaled down to either 3.7 or 1.2 MHz. However, it is shown that the system can also perform at higher frequencies if required by an application. The longest path (most timing critical) in the design, has a timing slack at 832 ps, which is equal to 35% of the clock period. The synthesis results are provided in Table 7.2. As mentioned in the previous section, the DSP was not synthesized, however an estimation of the area is obtained from Tensilica.

Table 7.2: Area obtained from the synthesis results.

Unit	**Area [mm^2]**	**% of total area**
DSP (incl. iram & dram)	0.106	15.5
Accelerator	0.022	3.2
$*$ (RALUT)	0.313×10^{-3}	-
Memory units	0.558	81.3
Total	0.686	100

Although the required memory storage was reduced with more than 75% by optimization techniques, it is expected with 256 kilobytes of memory in the design, that the memory units accounts for the largest part of the area, which is 81.3% of the total area. The DSP takes up 15.5% of the total area and is five times larger than the accelerator, taking up 3.2%. It should be noted that the DSP has 4 kilobytes or available memory, allowing for added functionality. For the RALUT implementation of the hyperbolic tangent function, the area is 313 μm^2.

2.2 Performance

To evaluate the performance of the design, the execution time is calculated for each functional unit together with the total execution time for the full RNN model. For this calculation the number of clock cycles is obtained from

the simulation and an operation frequency at 3.7 MHz ($\approx 0.27\,\mu s$) is used. The results are presented in Table 7.3

Table 7.3: Performance for DSP with accelerator at 3.7 MHz.

Function name	**Clock cycles**	**Exe Time [ms]**
GRU 256 inputs	9,510	2.57
∗ Gate U and R	6,159	1.66
∗ Memory cell	8,216	2.22
∗ RALUT	1	0.27×10^{-6}
∗ Output stage	1,293	0.35
GRU 128 inputs	6,224	1.68
Batch Normalization	3,712	1.00
Fully Connected	896	0.24
Sigmoid function	35	9.45×10^{-3}
Full sized model	24,089	6.50

For the full sized model the total execution time (for one classification) is 6.5 ms, which allows for real-time processing of 256 inputs. This provides a throughput of 153 classifications per second. Also, it is noted that there is a balanced workload between the DSP and the accelerator, with 43.1% of the time used in the DSP and 56.9% in the accelerator.

By scaling down the frequency to 1.5 MHz, the total execution time is 16.92 ms, which is fast enough for real-time processing of 40 inputs, which accounts for a complete set of MFCC coefficients.

2.3 Power and Energy Consumption

For the DSP, where synthesis was not possible, the estimated power consumption is obtained from Tensilica. This value is not an accurate estimation, as it does not reflect the real switching activity nor the power consumption caused by the parasitic capacitance and resistance in the design. However, to get an indication of how much the DSP and the accelerator contributes to the total power consumption, the value obtained in Tensilica is compared to the estimated power consumption for the accelerator. This estimation is obtained from the synthesis data and is also lacking the real switching activity[2] and the parasitic element. The numbers are presented in Table 7.4. As it is observed, the DSP accounts for more than half of the

[2]In the synthesis tool a switching activity of 0.02 toggle/ns is used.

Table 7.4: Power estimation with 0.9 V_{DD} and 430 MHz.

	$\mathbf{P_{leak}}$ **[mW]**	$\mathbf{P_{dyn}}$ **[mW]**	$\mathbf{P_{tot}}$ **[mW]**
DSP (incl. iram and dram)	N/A	N/A	18.75
Accelerator	0.39	15.64	16.03
Total system power	-	-	34.78

total power consumption. In order to obtain a more accurate estimation, the switching activity as well as data for the parasitic elements, must be included.

For input data represented by the MFCC, the frequency can be scaled down to 1.5 Mhz and the supply voltage to 0.7 V. This results in a reduced the power consumption, as presented in Table 7.5. It is noted that the

Table 7.5: Power estimation when scaling V_{DD} to 0.7 V and the frequency to 1.5 MHz.

	$\mathbf{P_{leak}}$ **[mW]**	$\mathbf{P_{dyn}}$ **[mW]**	$\mathbf{P_{tot}}$ **[mW]**
DSP (incl. iram and dram)	N/A	N/A	0.184
Accelerator	0.302	0.033	0.335
Total system power	-	-	0.519

leakage power is now dominating part of the total power consumption. As the synthesis was carried out using low voltage threshold (LVT) devices, decreasing the leakage power can be attempted, by configuring the synthesis tool to use regular threshold voltage devices and only apply low threshold voltage devices in timing critical paths.

For an operating frequency at 3.7 MHz and with a supply voltage at 0.7 V, the total power consumption is estimated to 636 μW. Lastly, the estimated energy consumption per classification is calculated to:

$$E_{per\ class} \approx 4.13\ \mu J/class \qquad @\,3.7MHz$$

$$E_{per\ class} \approx 8.78\ \mu J/class \qquad @\,1.5MHz$$

3 Design Comparison

Table 7.6 summarizes important metrics for this and related work.

Table 7.6: Performance comparison.

	This Work	**Xtensa Fusion F1**	**DNPU [8]**	**Laika [5]**
Technology	28nm	28nm	65nm	65nm
Algorithm	GRU	GRU	CNN & RNN	LSTM
Processor	Opt. DSP	DSP	RISC	-
Real-time processing	Yes	Yes	-	Yes
On-chip memory [kB]	252	252	290	32
Area [mm^2]	0.686	0.730	16	1.035
Voltage [V]	0.7	0.9	0.77	0.575
Frequency [MHz]	3.7	200	50	0.250
Power [mW]	0.636	9.42	34.6	0.005
Energy per Classification [μJ]	6.33	68.77	-	0.06

As the area and power consumption are estimated values for our work and the Xtensa Fusion F1, it is valid to do a comparison of these systems. It is seen that by introducing parallel execution we were able to reduce the frequency and supply voltage and by that achieve a lower power and energy consumption. A smaller area is also obtained. By scaling up the frequency our design is able to also process GRU models that requires higher performance.

The DNPU [8] has a high level of programmability, and can do inference of both CNN and RNN networks. The trade-off for this is a large area at 16 mm^2 and a high power consumption at 34.6 mW.

In Laika [5] they were able to achieve a very low power consumption at 5μW. This was possible due to a pure ASIC design that operates at 250 kHz and a low supply voltage at 0.575 V, while a 0.7 V supply is used for the SRAM units. [5]. Also, the design is limited to a maximum frequency at 8 MHz and with 32 kilobytes, the memory space does not allow for inference of larger networks. The level of reconfigurable options is limited.

Chapter 8

Conclusion

With the ongoing technological advancement within the area of machine learning together with the increasing number of handheld devices and IoT implementing this technology, the need for programmable hardware that is power and memory efficient is larger than ever.

In this thesis we sought to exploit optimization techniques in order to design a scalable and programmable low power hardware unit, for the inference of recurrent neural networks that are based on the GRU.

We have proposed an architecture consisting of a DSP integrated with an accelerator. A high level of programmability is obtained by including a DSP in the design. The DSP is optimized for low power consumption and we reduce the processing time, by creating application specific instructions. With the accelerator, parallelism is exploited to get rid of the computational bottlenecks within the GRU. This allows for lowering the clock frequency and by that a lower power consumption is achieved. Also, by implementing parallel computing of algorithms where the same input is used, the memory access is reduced. This helps reducing the power consumption of the memory units.

To deal with the high memory space required of the gated recurrent units, we make use of fixed point number representation together with quantization of the GRU parameters. This allows us to store the parameters using 8-bit.

The GRU includes the computational heavy activation functions sigmoid and hyperbolic tangent. To reduce the computation time of these, a polynomial approximation is used for the sigmoid while a RALUT is implemented for hyperbolic tangent.

In conclusion, a 75% reduction of the required memory space is achieved with an energy consumption of 4.13 μJ per classification, for real time processing of inputs sampled at a rate of 16,000 samples per second. In 28nm technology, the area is 0.686 mm^2. The operation frequency can be scaled up to 430 MHz, which also allows for the inference of GRU models that demands a higher performance.

Due to software complications, we were not able to retrieve the RTL files for the DSP. As a result, synthesis and layout of the complete system was not carried out. Resolving this issue as part of future work, would allow for a more accurate estimation of power consumption, by including the data of the parasitic components and also the switching activity for real data input. Also, by configuring the synthesis tool to implement high threshold voltage devices in paths that are not timing critical, a lower leakage voltage can potentially be achieved.

Through our study we have identified several methods that can be further researched for potentially reducing the required memory space. The quantization of weights can be exploited to obtained an even smaller word length e.g. 4-bit. However retraining of the network is then required. Another method is pruning, where redundant weights can be removed from the network [2]. Also, we see the technique called quantization aware training, as an interesting method to further explore. With this method the weights are quantized during network training.

Although, the reference RNN model used in this thesis, utilize a hard sigmoid function, other models might benefit from a lower error rate than what is obtained with the hard sigmoid approximation. A solution can be to implement a RALUT for the sigmoid function.

Lastly, for a more complete SoC design, the possibility for implementing the computations of the MFCCs into the DSP, can be explored.

Bibliography

[1] Hart and L. Hart, *How The Brain Works*. Basic Books, 1975.

[2] V. Sze, Y. H. Chen, T. J. Yang, and J. S. Emer, "Efficient Processing of Deep Neural Networks: A Tutorial and Survey," *Proceedings of the IEEE*, vol. 105, no. 12, pp. 2295–2329, 2017.

[3] W. Wahby, T. Sarvey, H. Sharma, H. Esmaeilzadeh, and M. S. Bakir, "The impact of 3D stacking on GPU-accelerated deep neural networks: An experimental study," *2016 IEEE International 3D Systems Integration Conference, 3DIC 2016*, 2017.

[4] L. Van Hee, R. Van Den Heuvel, T. Verheyden, and D. Baert, "Google employees are eavesdropping, even in your living room, VRT NWS has discovered." `https://www.vrt.be/vrtnws/en/2019/07/10/google-employees-are-eavesdropping-even-in-flemish-living-rooms`. Accessed: 2019-07-17.

[5] J. S. Giraldo and M. Verhelst, "Laika: A 5uW Programmable LSTM Accelerator for Always-on Keyword Spotting in 65nm CMOS," *ESSCIRC 2018 - IEEE 44th European Solid State Circuits Conference*, pp. 230–233, 2018.

[6] J. Chung, C. Gulcehre, K. Cho, and Y. Bengio, "Empirical Evaluation of Gated Recurrent Neural Networks on Sequence Modeling," pp. 1–9, 2014.

[7] S. Khandelwal, B. Lecouteux, L. Besacier, C. Gru, B. Lecouteux, and L. Besacier, "COMPARING GRU AND LSTM FOR AUTOMATIC

SPEECH RECOGNITION Shubham Khandelwal , Benjamin Lecouteux , Laurent Besacier To cite this version : HAL Id : hal-01633254," 2017.

[8] S. Dongjoo, L. Jinmook, L. Jinsu, L. Juhyoung, and Y. Hoi-Jun, "DNPU: An energy-efficient deep-learning processor with heterogeneous multi-core architecture," *IEEE Micro*, vol. 38, no. 5, pp. 85–93, 2018.

[9] "OCEAN: An on-chip incremental-learning enhanced processor with gated recurrent neural network accelerators," *ESSCIRC 2017 - 43rd IEEE European Solid State Circuits Conference*, pp. 259–262, 2017.

[10] Y. Lecun, Y. Bengio, and G. Hinton, "Deep learning," may 2015.

[11] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016. `http://www.deeplearningbook.org`.

[12] K. Cho, B. Van Merriënboer, C. Gulcehre, D. Bahdanau, F. Bougares, H. Schwenk, and Y. Bengio, "Learning phrase representations using RNN encoder-decoder for statistical machine translation," in *EMNLP 2014 - 2014 Conference on Empirical Methods in Natural Language Processing, Proceedings of the Conference*, pp. 1724–1734, 2014.

[13] A. Graves and J. Schmidhuber, "Framewise phoneme classification with bidirectional lstm and other neural network architectures," *Neural Networks*, vol. 18, no. 5, pp. 602 – 610, 2005. IJCNN 2005.

[14] K. Greff, R. K. Srivastava, J. Koutnik, B. R. Steunebrink, and J. Schmidhuber, "LSTM: A Search Space Odyssey," *IEEE Transactions on Neural Networks and Learning Systems*, vol. 28, no. 10, pp. 2222–2232, 2017.

[15] Y. Goldberg, *Yoav Goldberg Book - Neural Network Methods in Natural Language Processing*. 2017.

[16] L. Rabiner and B. Juang, "An introduction to hidden markov models," *IEEE ASSP Magazine*, vol. 3, no. 1, pp. 4–16, 1986.

[17] T. Ganchev, N. Fakotakis, and G. Kokkinakis, "Comparative Evaluation of Various MFCC Implementations on the Speaker Verification Task," *October*, vol. 1, no. 3, pp. 191–194, 2005.

[18] J. Donahue, L. A. Hendricks, S. Guadarrama, M. Rohrbach, S. Venugopalan, K. Saenko, and T. Darrell, "Long-term recurrent convolutional networks for visual recognition and description," *CoRR*, vol. abs/1411.4389, 2014.

[19] Z. Zhang, D. Robinson, and J. Tepper, "Detecting hate speech on twitter using a convolution-gru based deep neural network," 03 2018.

[20] T. Mikolov, K. Chen, G. Corrado, and J. Dean, "Efficient estimation of word representations in vector space," *1st International Conference on Learning Representations, ICLR 2013 - Workshop Track Proceedings*, pp. 1–12, 2013.

[21] S. Ioffe and C. Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift," *32nd International Conference on Machine Learning, ICML 2015*, vol. 1, pp. 448–456, 2015.

[22] J. Rabaey, *Digital Integrated Circuits: A Design Perspective*. Prentice Hall electronics and VLSI series, Prentice Hall, 1996.

[23] P. Marwedel, *Embedded System Design*. Embedded Systems, Springer International Publishing, 3 ed., 2018.

[24] Cadence Design Systems, Inc., *Tensilica Fusion F1 DSP - Configurable processor for Internet of Things applications*, 2016.

[25] S. Kapur, A. Mishra, and D. Marr, "Low Precision RNNs: Quantizing RNNs Without Losing Accuracy," pp. 1–5, 2017.

[26] R. Krishnamoorthi, "Quantizing deep convolutional networks for efficient inference: A whitepaper," 2018.

[27] A. H. Namin, K. Leboeuf, R. Muscedere, H. Wu, and M. Ahmadi, "Efficient hardware implementation of the hyperbolic tangent sigmoid function," *Proceedings - IEEE International Symposium on Circuits and Systems*, no. June, pp. 2117–2120, 2009.

[28] K. Leboeuf, A. H. Namin, R. Muscedere, H. Wu, and M. Ahmadi, "High speed VLSI implementation of the hyperbolic tangent sigmoid function," *Proceedings - 3rd International Conference on Convergence and Hybrid Information Technology, ICCIT 2008*, vol. 1, no. November, pp. 1070–1073, 2008.

[29] M. Al-Nsour and H. S. Abdel-Aty-zohdy, "Implementation of programmable digital sigmoid function circuit for neuro-computing," *Midwest Symposium on Circuits and Systems*, pp. 571–574, 1998.

[30] Patterson, *Computer Architecture A Quantitative Approach.* Springer International Publishing, 6th ed., 2019.

[31] *Xtensa LX7 Microprocessor*, data book ed., December 2019.

[32] Y.-H. Chen, T.-J. Yang, J. Emer, and V. Sze, "Understanding the limitations of existing energy-efficient design approaches for deep neural networks," *SysML Conference (SYSML'18)*, vol. 2, no. L1, p. L3, 2018.

[33] "Machine learning crash course." `https://developers.google.com/machine-learning/crash-course/classification/accuracy?hl=ru`. Accessed: 2020-06-01.

Chapter A

Accuracy and class-imbalanced data set

Accuracy is a metric for evaluating classification models and is defined as follows [33],

$$Accuracy = \frac{Numberofcorrectpredictions}{Totalnumberofpredictions}$$

For binary classification, accuracy can also be calculated in terms of positives and negatives as follows [33],

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN}$$

TP: True Positives - (model predicted correct, positive/1)
TN: True Negatives - (model predicted correct, negative/0)
FP: False Positives - (model predicted positive, but reality is negative)
FN: False Negatives - (model predicted negative, but reality is positive)

However, accuracy alone doesn't tell the full story when you're working with a class-imbalanced data set, where there is a significant disparity between the number of positive and negative labels [33].
Two metrics better suited for evaluating class-imbalanced data set are precision and recall.

Precision attempts to answer the following question:
What proportion of positive identifications was actually correct [33].

$$Precision = \frac{TP}{TP + FP}$$

Recall attempts to answer the following question: What proportion of actual positives was identified correctly [33].

$$Recall = \frac{TP}{TP + FN}$$

To fully evaluate the effectiveness of a model, you must examine both precision and recall. Unfortunately, precision and recall are often in tension. That is, improving precision typically reduces recall and vice versa.

Chapter B

Graphical Representation of Activation Functions

Graphical representation of the activation functions discussed. Figure B.1 Sigmoid, Figure B.2 hyperbolic tangent, Figure B.3 ReLU.

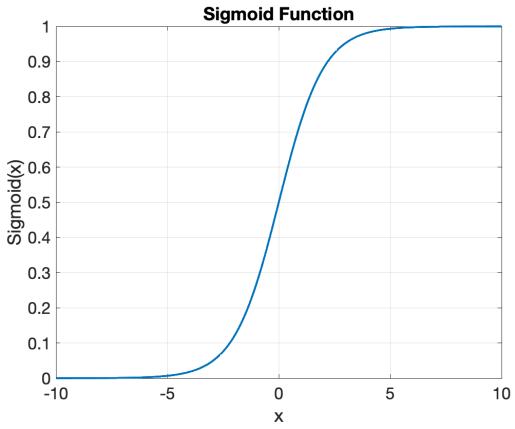

Figure B.1: Sigmoid function.

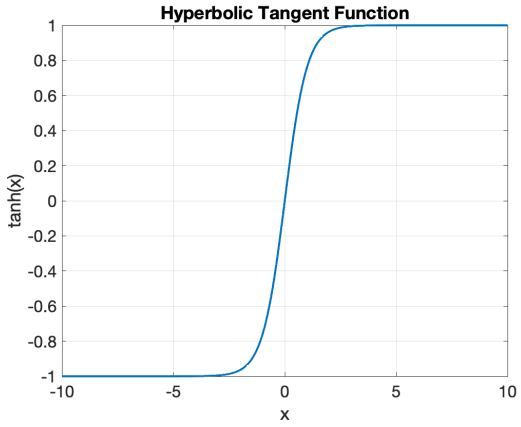

Figure B.2: Hyperbolic Tangent function.

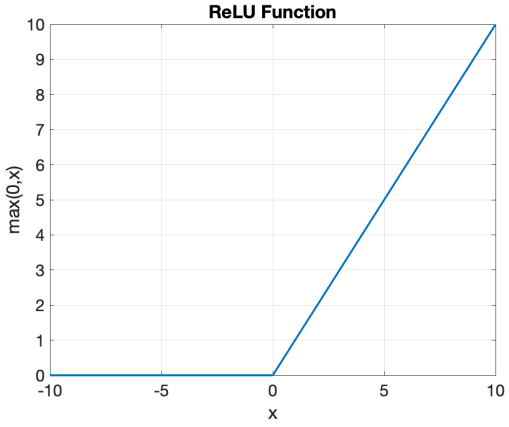

Figure B.3: ReLU function.

Printed by Tryckeriet i E-huset, Lund 2020

LUND
UNIVERSITY

Series of Master's theses
Department of Electrical and Information Technology
LU/LTH-EIT 2020-771
http://www.eit.lth.se