KU LEUVEN

FACULTEIT INGENIEURSWETENSCHAPPEN

Efficient Hardware Mapping of Long Short-Term Memory Neural Networks for Automatic Speech Recognition

Georgios N. Evangelopoulos

Thesis submitted for the degree of
Master of Science in
Electrical Engineering, option
Electronics and Integrated Circuits

Thesis supervisor:
Prof. dr. ir. Marian Verhelst

Assessors:
Prof. dr. ir. W. Dehaene
Prof. dr. ir.H. van Hamme

Mentor:
Ir. Bert Moons

Academic year 2015 – 2016

Preface

I would like to thank everybody who kept me busy this last semester, especially my supervisor Prof. dr. ir. Marian Verhelst, and my assistant, Ir. Bert Moons. Their support and feedback is what made this work possible. My sincere gratitude also goes to my family and friends who supported me in this journey.
I sincerely hope that this work improves, in its own small way, the lives of many.

Georgios N. Evangelopoulos

Contents

Abstract

Recently, Long Short-Term Memory Neural Networks (LSTMs) have become state-of-the-art models for a variety of machine learning tasks, including but not limited to Automatic Speech Recognition, video and action classification, unsegmented handwriting recognition and generation, polyphonic music modelling and protein secondary structure prediction. However, LSTMs are usually very memory and computationally intensive. In order to enable LSTM-based classification in embedded systems such as smartphones and ubiquitous electronics for wireless sensor networks and the internet of things, their energy footprint should be reduced.

This work examines how LSTMs behave as different methods of approximate computing are applied. Using shorter word lengths to represent architectural parameters and data, pruning LSTM connections and coarsely approximating the non-linear activation functions present in neural networks are some of the techniques applied. Our results show that LSTM neural networks allow for many algorithmic simplifications while suffering little to no penalty in their predictive performance. No loss in accuracy is observed when network parameters and input data are quantized to 8 bits, while allowing for a 4% relative accuracy loss allows for quantization to 7-bits. Allowing for that error margin, 70% of the LSTM connections can be pruned. Furthermore, coarsely approximating the nonlinear activation functions with simple look-up table approximations does not introduce a loss in accuracy even with tables containing only 4–8 entries.

These algorithmic-level techniques are combined with architectural modifications to an Application Specific Instruction Set Processor (ASIP) to reduce energy consumption in an LSTM-based Automatic Speech Recognition system. These modifications include parallelism in the processor data path, introduction of more pipeline stages, extensions to the Instruction Set Architecture and RTL-level modifications that can further reduce the switching activity of the processor. The algorithmic approximations along with the customized ASIP architecture can lead to 80% energy reduction due to quantization and up to 67% due to pruning. The LUT activation functions also allow for very inexpensive implementations that can lead to even larger energy savings.

We show that LSTMs behave well even when undergoing the simplest approximations and that when combined with a hardware architecture that can realize the predicted energy savings, they can be efficiently used for embedded sequential machine learning tasks on platforms with a limited energy budget.

List of Figures

List of Tables

List of Algorithms

List of Abbreviations

Abbreviations

ASR	Automatic Speech Recognition
LSTM	Long Short-Term Memory
NN	Neural Network
ASIP	Application Specific Instruction-set Processor
SIMD	Single Instruction Multiple Data
MFCC	Mel-Frequency Cepstral Coefficients
FC	Fully Connected (Neural Network)
RNN	Recurrent Neural Network
BLSTM	Bidirectional LSTM
CMOS	Complementary Metal-Oxide-Semiconductor
WL	Word Length
DRAM	Dynamic Random-access Memory
LUT	Look-up Table
FPGA	Field Programmable Gate Array
ASIC	Application Specific Integrated Circuit
ISA	Instruction Set Architecture
HDL	Hardware Description Language
ADL	Architecture Description Language
SISD	Single Instruction Single Data
ALU	Arithmetic Logic Unit
DVFS	Dynamic Voltage and Frequency Scaling

Chapter 1

Introduction

> The knowledge of external things which we derive from the indications of our senses is for the most part the result of inference.
>
> Lord Rayleigh

1.1 Knowledge and the Machine

Machine Learning, the field of study that grants computers the ability to learn without being explicitly programmed, has a long and exciting history. Relying on the Baconian method of inductivism, scientists have been advancing this Artificial Intelligence field with an ever-increasing rate, allowing for the development of machinery that seems to have the ability of thinking. This elusive quality of Artificial Intelligence systems has surprised and intrigued, while at the same time raised eyebrows and concerns.

On the one hand, engineers and scientists are deeply involved in the challenging quest of teaching computers how to think. This challenge is a vastly inter-disciplinary field, tied to mathematics, psychology, biology and (neuro)physiology, philosophy, logic, linguistics and so on, and this has led to highly diversified methods and techniques, each being influenced differently by the vast number of related fields.

On the other hand, philosophers and sociologists of science have also joined a less technical albeit equally important debate; What will be the societal and humanitarian significance of the advancement of such a field? Which tasks can be trusted to "intelligent" machines and which not? The diversity of results and opinions is remarkable. Yes, maybe when the arena is as well-defined and pure as a board game, artificial intelligence might seem superhuman and formidable. However, when the arena is ill-defined and its rules are consistently challenged by uncertainty, the machines are constantly underperforming and have not gained societal acceptance or trust.

As the debate continues, science and technology have not stood still; little by little, researchers push the frontier of their fields, as they have always done.

1.2 The Ubiquity of Electronics

The advances in the field of electronics are also quite remarkable. Aside from the computer and the smartphone and the ways they revolutionized thousands of aspects of every day life, electronics have become integrated with reality in countless other ways that are much less known yet equally significant. Advances in industrial automation, telecommunications, aerospace and medicine, just to name a few, have all been significantly aided by the existence and development of microelectronics. The trends of networking, data collection, automation, intelligent systems, and technology that can gather its own information are setting increasingly challenging goals for microelectronics, the backbone of the digital era.

Constraints that are currently arising in the field of microelectronics have much to do with the usage of less electrical power. The rapid, exponential growth of Very Large Scale Integration (VLSI) technology has been very favourable regarding what microelectronics can actually do; Energy storage technology (mainly batteries), on the other hand, hasn't enjoyed the same rate of growth. The imminent ubiquity of electronics into the mainstream is imposing stringent energy constraints for engineers and hardware designers. Researchers are feverishly seeking methods and design methodologies in various level of abstraction in order to battle the increase in power consumption that inevitably comes with increased functionality.

1.3 This work

This work is concerned with a topic that fuses the two disciplines mentioned above; Machine Learning and Low-Power Hardware Design. More specifically, it addresses the functionality of a machine learning system that performs Automatic Speech Recognition and its deployment on a custom hardware architecture tailored for the application in terms of energy consumption. Much related work exists around this fusion of fields, but the author believes that this work contributes some previously unaddressed insight and will, therefore, be useful to the community of fellow researchers.

This work is structured as follows: In Chapter 2, related work in the field of Automatic Speech Recognition is explored, recent trends are presented and finally Long Short-Term Memory (LSTM) neural networks are identified as a promising, well-performing candidate for Automatic Speech Recognition.

Chapter 3 presents the linguistic data that will be used for the development and evaluation of an Automatic Speech Recognition LSTM system, along with the processing that is necessary for their efficient usage.

Chapter 4 presents Long Short Term Memory neural networks and the overall ASR architecture that incorporates LSTMs. It also discusses the implementation of the overall architecture in the Torch machine learning framework and the training and testing procedures. Finally, it presents the system's overall functionality and reports results on its performance.

Chapter 5 investigates the architecture of Chapter 4 from a point of view that is sensitive to energy consumption; The energy requirements of LSTMs and generally Neural Network approaches to machine learning tasks are discussed. Several methods that can be applied in the algorithmic level in order to reduce the energy consumption of the system are presented. The system's performance is assessed with respect to the applied energy-saving optimizations.

Chapter 6 describes a modified Application Specific Instruction-set Processor (ASIP) architecture that exploits the aforementioned algorithmic optimizations. The ASIP architecture is tailored to the needs of the LSTM-based machine learning architecture and its energy consumption is modelled based on its predicted functionality.

Finally, Chapter 7 concludes this work. The motivation for low-power, LSTM-based machine learning system are restated and the contributions made in this work and the achieved results are summarized.

Chapter 2

Automatic Speech Recognition

For centuries, engineers and scientists have been intrigued by the concept of human-mimicking machines, and speech production and recognition are not exceptions. Attempts at designing systems that understand spoken languages were made since at beginning of the 20th century and ever since, more and more accurate systems have been developed. To understand the challenges that exist in the creation of such systems, it is important to understand the basic concepts of *sound* and *speech* as well as understand the advances and the backtracks that have matured the discipline of Automatic Speech Recognition (ASR).

2.1 Sound

Sound, the stimulus to auditory perception, is defined as "oscillation in pressure, stress, particle displacement, particle velocity etc., propagated in a medium with internal forces, or the superposition of such propagated oscillation"[2].

In the author's opinion, however, this mere technical definition cannot capture the magnificent nature of this phenomenon. Sound can carry valuable information; It can induce strong feelings; It can shape our perspective of the world and signal events that would otherwise be completely undetectable with any other tactile sense.

Driven by sound's significance, humans have always longed to understand sound waves and the possibilities that emerge from their manipulation; Their applications in art, medicine, biology, physics and engineering are of indisputable importance.

It is easy, however, to overlook one of the most significant applications of sound, one that has a form so intuitive, that humans tend to forget that this is what enabled them to leap forward, form societies, create civilization and advance considerably faster than other creatures: Speech.

2.2 Speech

Speech is the basic tool for oral communication. Using speech and language, people can convey their ideas effectively, by either communicating short and specific messages, or by painting big, complex mental images. It is vocalized communication based

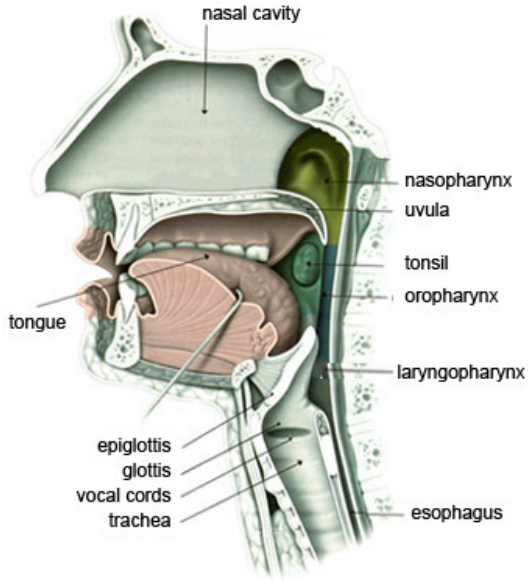


Figure 2.1: The vocal apparatus. The vocal apparatus is the mechanism that allows humans to produce speech. It consists of the lungs, the vocal cords and a series of resonant chambers, like the pharynx, the mouth and the nasal cavities.

upon the syntactic combination of lexicals and names that are drawn from very large vocabularies. Each spoken word consists of phonetic combinations of a limited set of vowel and consonant speech sound units, also called *phones*[6].

The *production of speech* is indeed a very big chapter in the field of linguistics and phonetics; it involves the selection of words, the creation and organization of the appropriate grammatical forms and then the articulation of the resulting sounds using the vocal apparatus. The vocal apparatus (Figure 2.1) consists of the human nose, throat and vocal chords, and allows for the production of many unique sounds[10].

The *perception of speech* is the process of intepreting and understanding the sounds used during the production of speech[28]. Speech perception is closely tied to the field of linguistics (especially phonetics) and cognitive psychology. Research on these fields seeks to understand the methods that humans follow in order to *recognize* incoming sound waves as speech signals, the underlying rules used to *decode* them into words and finally, the procedures followed to *extract meaning* and *information*. It is noteworthy that humans, in contrast to machines, can, in most cases, effortlessly understand speech regardless of speaker variations (pronunciation, dialect voice timbre) and environmental factors (e.g. background noise or discontinuities in the speech stream).

2.3 Advances in Automatic Speech Recognition

Since the emergence of Speech Recognition as a research field, hundreds of different approaches have been taken and thousands of models have been designed to face the

task of recognizing human speech:

- As early as 1952, Davis, Biddulph, and Balashek of Bell Laboratories built a system for isolated digit recognition for a single speaker [7], using characteristics of the power spectrum of the speech signal during vowel regions of each digit. This work implicitly assumed that each utterance contained exactly one digit, and therefore didn't use any technique to "segment" the speech signal into smaller parts that would each contain one recognizable "entity" (be a digit, a vowel, a phone, or even a whole word).

- In the 1960's, laboratories in Japan developed systems that tackled the speech recognition task. At the Radio Research Lab in Tokyo, Sakai and Doshita [29] first introduced a speech segmenting module that helped in the analysis of different parts of the input speech signal.

- By 1970, scientists and engineers started formulating the basic ideas of applying pattern recognition technology to speech recognition. The first real ASR product, the VIP-100 system influenced the U.S. Department of Defense and its Advanced Research Projects Agency (ARPA) to fund the Speech Understanding

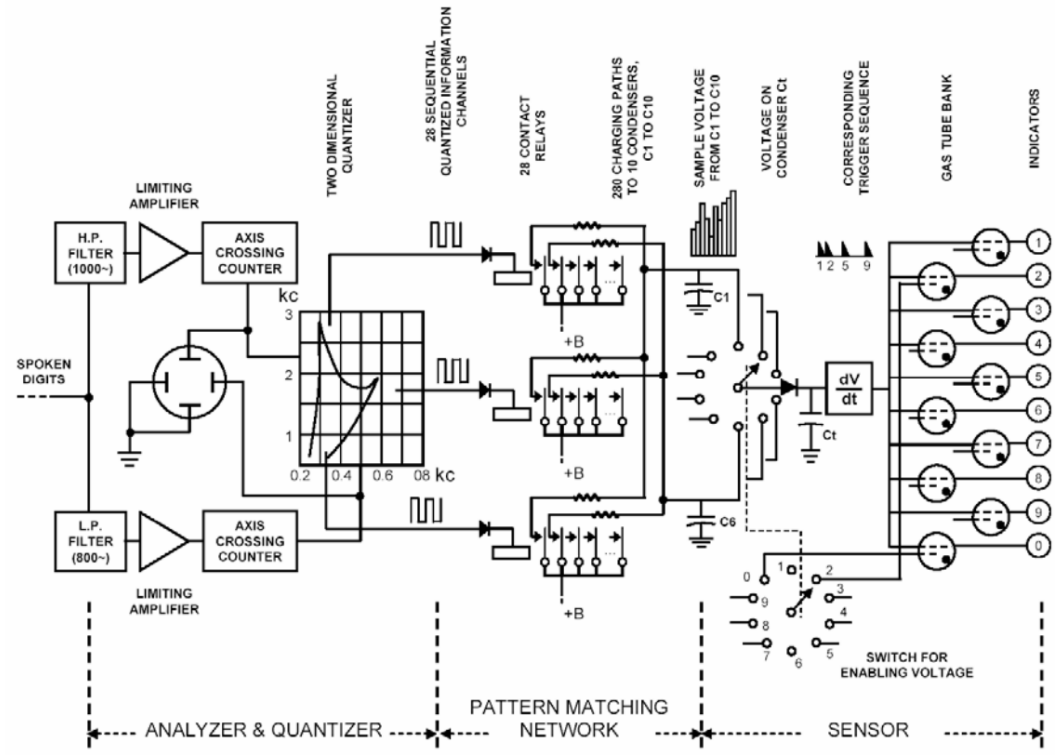


Figure 2.2: The first speech recognizer, made in 1952. Researchers at Bell labs devised a circuit that discriminated between digits by using the formant frequencies measured during the vowel regions of each digit.

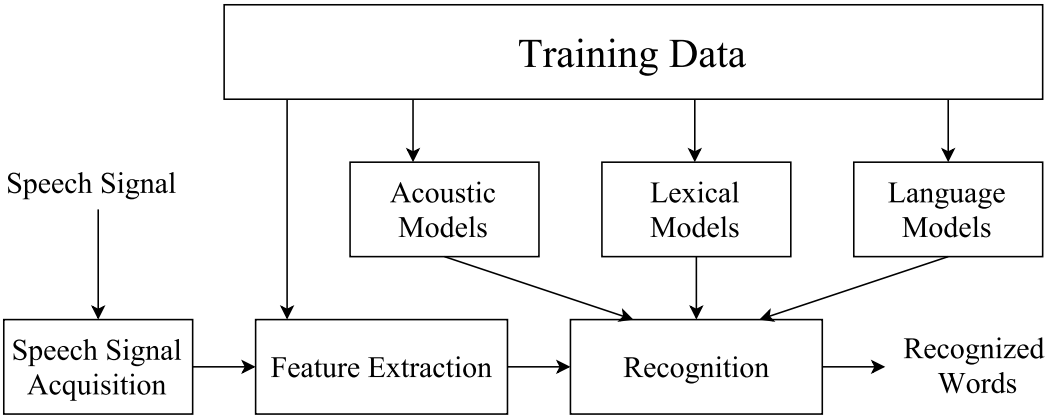


Figure 2.3: A statistical approach to speech recognition. Based on training data and expert knowledge, acoustic, lexical and language models take part in the recognition of words in a speech signal.

Research (SUR) program in the beginning of the decade. The system "Harpy" developed in Carnegie Mellon University [24] was one of the first to use graph search and finite state networks.

- In the 1980's, advances in statistical modelling of speech (most notably Hidden Markov Models, HMM) allowed a whole new dimension in ASR. Instead of using intuitive approaches, more concrete statistical modelling was introduced. Common knowledge is that not all sequences of sounds are "equiprobable" in the context of a given language. Some are more common than others, other are less probable and some others are downright impossible. Using datasets of spoken utterances annotated at the word level, statistical models that estimate the probability distribution of a sequence of words occurring within the context of a specific language were defined and used.

The approaches mentioned above share a common value. They require many hand-designed, ad-hoc strategies to deal with many "quirks" present in the task of ASR, such as the variation of speech signals occurring for the same utterance when spoken by different persons or the fact that speech contains strong short-term (or maybe even long term) temporal features. As a result, the overall models tend to be complicated and heterogeneous, having many modules interfacing with each other in non-trivial ways.

2.4 Neural Networks in ASR

In the 1990's, scientists started recognizing the fact that a probabilistic approach to speech recognition would be inapplicable as the requirements of the task grew. Speech recognition was transformed into an optimization problem, and neural networks were suggested as a solution [20].

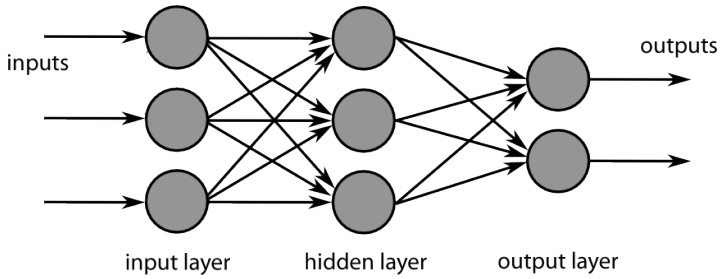


Figure 2.4: A feed-forward neural network as a classifier. Each layer is fully connected with the next. Each connection carries a weight w, and the outputs of each layer are passed through a nonlinear activation function before they are propagated to the next layer.

2.4.1 Feed-forward Neural Networks

Feed-forward neural networks are models that are used to approximate unknown functions that have a large number of inputs. They can be thought of as systems with interconnected neurons that have weighted inputs, where the weights can be tuned by acquired experience. The output of a fully connected feed-forward neural network with one layer, a N-dimensional input $\mathbf{x}$ and an M-dimensional output $\mathbf{y}$ can be seen in equation 2.1.

$$\mathbf{y} = g(\mathbf{W}\mathbf{x} + \mathbf{b}) \tag{2.1}$$

where:

$\mathbf{x}$ = an N-dimensional input vector
$\mathbf{y}$ = an M-dimensional output vector
$\mathbf{W}$ = the $M \times N$ weight matrix
$\mathbf{b}$ = the M-dimensional bias vector
$g(\cdot)$ = a nonlinear activation function.

In order however for neural network systems to be able to accurately model the unknown functions they're intended to, they have to be *trained*, i.e. be presented with an adequate number of input-output pairs. With a non-trivial and computationally expensive procedure called *backpropagation*, the input-output pairs can be used to tune the weights of the connections between neurons so that a loss function is minimized. The process of training neural networks is a vast research field and further analysis falls out of the scope of this work.

This neural network approach addressed, in a way, the problem of the increased complexity of the designed models mentioned above, but still, it did not face the fact that *temporal context* is of paramount importance to understanding the sequences of sounds spoken by a human being. This is due to the fact that neural networks have a static input layer; they do not treat feature vector sequences at their inputs as if they were temporally correlated, but instead, assume no specific "order" in the data that they process.

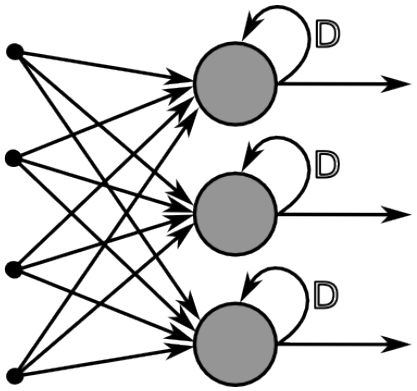


Figure 2.5: A Recurrent Neural Network (RNN). In RNNs, the output of each layer is connected to its input, creating an internal *state* that can, in theory, model temporal dependencies among inputs.

2.4.2 Recurrent Neural Networks

Until very recently, in the field of neural networks, researchers have been trying to incorporate temporal context into their models by manually designing "workarounds". One widely used "workaround" is using buffered inputs, i.e. expanding the number of inputs to the network to accommodate a longer series as a single input, effectively allowing the network to see the input data through a wider time window[1]. However, most of these approaches assume that the context inside this fixed-width window is enough for correct classification and in turn imply, even more absurdly, that the context outside the window is of no importance whatsoever. Other similar handcrafted designs include streaming inputs and the sliding window architecture[26]. It is easy to see that these approaches suffer from the general nuances of models with hand-designed features: They make strong assumptions and they do not scale well.

An effort to natively incorporate temporal dependencies into neural networks led to the development of Recurrent Neural Networks (RNNs). In these recurrent architectures, the layer outputs are fed back as part of the input; this creates an internal state in the network and allows it to model, to some extent, temporal variations. RNNs (see Figure 2.5) can, theoretically, model a plethora of tasks that have sequential inputs.

The output of a simple recurrent neural network with one layer, a N-dimensional input $\mathbf{x}$ and an M-dimensional output $\mathbf{h}$ can be seen in equation 2.2.

$$\mathbf{h_t} = g(\mathbf{W}\mathbf{x_t} + \mathbf{R}\mathbf{h_{t-1}} + \mathbf{b}) \tag{2.2}$$

where:

$\mathbf{x}$ = an N-dimensional input vector
$\mathbf{h}$ = an M-dimensional output vector
$\mathbf{W}$ = the $M \times N$ feed-forward weight matrix
$\mathbf{R}$ = the $M \times M$ recurrent weight matrix
$\mathbf{b}$ = the M-dimensional bias vector
$g(\cdot)$ = a nonlinear activation function.

In contrast to Feed-forward NNs, RNNs need to be presented with *sequences* of input-output pairs (i.e. the order of the examples presented to the network *matters*). Their main disadvantage, however, is that these networks have proven to be very difficult to train[27]; The backpropagated error gradients vanish or explode very fast with respect to the time lag between two important events. Due to that fact, their usage and success in ASR and other related tasks has been limited.

2.5 LSTM: Long Short-Term Memory

Long Short-Term Memory (LSTM) [17] is a modified RNN architecture that can model arbitrarily long time dependencies and is currently regarded as state-of-the-art for coping with sequential machine learning tasks such as unsegmented handwriting recognition[13], image tagging[3], video recognition[31] and obviously, Automatic Speech Recognition. It was originally proposed by Hochreiter and Schmidhuber as a solution to the vanishing gradient problem in RNN architectures. The rest of this work will address Long Short-Term Memory. In the following chapters, the data used to train LSTM models and an LSTM Automatic Speech Recognition architecture will be presented.

2.6 Conclusion

Automatic Speech Recognition has taken big steps since its first inception in the beginning of the 20th century. Ad-hoc, intuitive approaches were followed in the '50s and '60s, while pattern recognition and statistical modelling techniques emerged in the '70s and the '80s. Later on, speech recognition transformed into an optimization problem and neural networks were suggested as modelling tools, but were soon abandoned due to the fact that they had no way of incorporating temporal dependencies into their learning procedure, a crucial point in speech recognition. Recurrent Neural Networks that can model temporal context were proposed as a solution, but their difficulties in training hindered their adoption.

Currently, Long Short-Term Memory (LSTM) neural networks are considered state-of-the-art for speech recognition and are used widely both in academia and in industry, achieving record-breaking results. In Chapter 3, the data used to train LSTM models is presented, while in Chapter 4 an LSTM architecture for Automatic Speech Recognition is trained and evaluated.

Chapter 3

The Automatic Speech Recognition Dataset

3.1 Introduction

In order to build robust acoustic models, applying good training algorithms to a suitable set of acoustic data is of paramount importance. The better the quality and nature of the data, the better the quality and nature of the learning procedure and the bigger the chances that the trained acoustic model will reveal its true potential. The procedure of training a machine learning model using an existing dataset can be allegorized with a teacher passing on her knowledge to a new, promising student; The more knowledgeable the teacher is and the better she is at spelling out what's necessary, the more the student is able to learn and let his talent unravel.

Hence, the success of the training algorithms is highly dependent on the quality and details of the annotation of the data they are trained on. Many datasets, however, are insufficiently annotated and only a few of them include labels at the phone level, due to the tedious, lengthy and manual procedure of doing so. In the context of speech recognition, a dataset with complete phonetic transcriptions of its audio data is invaluable.

In this work, the TIMIT Dataset [9] is used as the training material of the acoustic models. The motivation for its usage originates from the following points:

- It is fully and manually annotated at the phone level;
- It is of manageable size, yet considered thorough and complete, allowing for model development within reasonable time frames;
- A license for its usage has been obtained in previous research projects;
- It is widely used by the speech recognition research community, allowing for direct comparisons with the literature and quantifiable, verifiable results.

Sentence Type	Sentences	Speakers	Total	Sentences per Speaker
Dialect (SA)	2	63	1,260	2
Compact (SX)	450	7	3,150	5
Diverse (SI)	1,890	1	1,890	3
Total	2,342		6,300	10

Table 3.1: The material of the TIMIT dataset

3.2 The TIMIT Dataset

The DARPA TIMIT Acoustic-Phonetic Continuous Speech Corpus (TIMIT - Texas Instruments (TI) and Massachusetts Institute of Technology (MIT)) [9], described in [36], contains recordings of phonetically-balanced prompted English speech. With a total of 6300 sentences (10 sentences spoken by each of 630 speakers from 8 major dialect regions of the United States) the total speech time in TIMIT amounts to 5.4 hours. The sentences present in the dataset are manually annotated at the phone level in a timestamped fashion that allows for the exact positioning and duration of each phone during a sentence.

TIMIT consists of 2342 distinct prompts: 2 dialect sentences (SA), 450 phonetically compact sentences (SX) and 1890 phonetically diverse sentences (SI). The two SA sentences are read by all 630 speakers, since they are meant to expose dialectal variants. The 450 SX sentences were read by 7 different speakers, while the 1890 SI sentences were read only once by the 630 speakers. Table 3.1 better summarizes the TIMIT dataset's speech material.

The TIMIT Corpus documentation suggests training (70%) and test sets, as described in Table 1. The training set contains 4620 utterances, but in this work and according to common practice found in the literature, the dialect sentences (SA) are removed, thus reducing the training set to 3696 utterances from 462 speakers. The test set contains 1344 utterances from 168 speakers. In this work, and again in accordance to the literature, the core test set is used, which is an abridged version of the complete testing set consisting of 192 utterances. Moreover, a validation set of 400 sentences is created in line with Halberstadt [16]. The division can be seen in Table 3.2. The training and test sets do not overlap.

The phonetic alphabet used to transcribe the TIMIT utterances consists of 61 phones and is presented in Table 3.3.

Set	Speakers	Sentences	Hours
Training	462	3,696	3.14
Core test	24	192	0.16
Complete test set	168	1,344	0.81

Table 3.2: The division of the TIMIT dataset

Index	Phone Label	Example	Index	Phone Label	Example
1	iy	b*ee*t	32	z	*z*one
2	ih	b*i*t	33	zh	a*z*ure
3	eh	b*e*t	34	f	*f*in
4	ey	b*ai*t	35	th	*th*in
5	ae	b*a*t	36	v	*v*an
6	aa	b*o*b	37	dh	*th*en
7	aw	b*ou*t	38	m	*mom*
8	ay	b*i*te	39	n	*noon*
9	ah	b*u*t	40	ng	si*ng*
10	ao	b*ough*t	41	em	bott*om*
11	oy	b*oy*	42	nx	wi*nn*er
12	ow	b*oa*t	43	en	butt*on*
13	uh	b*oo*k	44	eng	Wash*ing*ton
14	uw	b*oo*t	45	l	*l*ay
15	ux	t*oo*t	46	r	*r*ay
16	er	b*ir*d	47	w	*w*ay
17	ax	*a*bout	48	y	*y*acht
18	ix	deb*i*t	49	hh	*h*ay
19	axr	butt*er*	50	hv	a*h*ead
20	ax-h	s*u*spect	51	el	bott*le*
21	jh	*j*oke	52	bcl	b closure
22	ch	*ch*oke	53	dcl	d closure
23	b	*b*ee	54	gcl	g closure
24	d	*d*ay	55	pcl	p closure
25	g	*g*ay	56	tcl	t closure
26	p	*p*ea	57	kcl	k closure
27	t	*t*ea	58	q	glotal stop
28	k	*k*ey	59	pau	pause
29	dx	mu*dd*y	60	epi	epenthetic silence
30	s	*s*ea	61	h	begin/end marker
31	sh	*sh*e			

Table 3.3: The phonetic alphabet of TIMIT

This phonetic alphabet is sometimes considered too fine-grained for practical use, and is sometimes collapsed into a set of 48 phones for training and into a set of 39 phones for testing purposes [22]. In this work, however, both in training and evaluation, the full set of 61 labels is used.

3.3 Preprocessing

In the field of machine learning, preprocessing of the available data before using them for training and testing purposes is almost always necessary. Preprocessing is a crucial step towards the creation of a robust model regardless of the task. The preparation of data can include feature extraction and selection, dimensionality reduction of high-dimensional data, normalization, cleaning, decorrelation, manipulations in the time or frequency domain etc.

3.3.1 Preprocessing in ASR

In the field of ASR, countless methods have been evolved with respect to the preprocessing of audio signals. In contrast with other pattern recognition tasks such as image or handwritten digit classification [21] (where the signal to be classified can be regarded as stationary over time) speech is a time-varying process and speech signals cannot be regarded as stationary over time. This observation proves to be a crucial point that influences not only the selection of preprocessing techniques that are to be applied but also the choice of a candidate model's architecture. The audio signal representing a single utterance (spoken sentence) can also vary significantly depending on the voice timbre, the speech rate and the pronunciation of the speaker, thus making it speaker-dependent. Background noise and other parameters also add to the variations in speech signals.

Performing Automatic Speech Recognition involves, to a large extent, the identification of the linguistic content in a speech signal. The temporal envelope of audio signals can, to some extent help identify linguistic content [30][8], but the behaviour of the vocal apparatus and "consequently" the spoken phones are better manifested in the frequency domain, in the short-time power spectrum [19]. Therefore, an appropriate extraction of features from the short-time power spectrum will play a major role in the training of any acoustic model.

The reasons that motivate the preprocessing of the TIMIT data before it is used in this work can be summarized as follows:

- The short-time power spectrum is a good feature for the recognition of linguistic content in a speech signal, while its temporal envelope is not;
- A canonical split of a speech signal into a sequence of discreet features is necessary for the training and testing the LSTM ASR architecture;
- Decorrelation of data aids classifiability;
- Standardization of data leads to better convergence;

- Alignment with the literature allows for comparable, quantifiable and easily interpretable results.

In this work, the preprocessing of the TIMIT data follows a well-documented and accepted method which will be briefly explained below.

3.3.2 MFCC extraction

The Mel Frequency Cepstral Coefficients (MFCC) is a feature extraction technique that has been used for years in Automatic Speech Recognition [25]. Although first mentioned by Bride and Brown in 1974, Paul Mermelstein developed it further based on experiments on the human misconception of words [23]. MFCC is an attempt to represent the linguistic message of a signal by incorporating facts about human speech perception. More specifically, it emphasizes the logarithmic perception of pitch and loudness and eliminates speaker dependent characteristics. The following steps describe MFCC extraction.

For a given one-dimensional discrete-time speech signal $s(n)$:

1. Split the signal in overlapping frames with duration of 25ms and 10ms frame shift;

2. For each speech frame, apply a windowing function and calculate the Discrete Fourier Transform;

$$h(n) = \alpha - \beta \cos\left(\frac{2\pi n}{N-1}\right) \tag{3.1}$$

$$S_i(k) = \sum_{n=1}^{N} s_i(n)h(n)e^{-j2\pi kn/N} \quad 1 \leq k \leq K \tag{3.2}$$

   where:

   i = the frame index
   $h(n)$ = an N sample long hamming window
   K = the length of the DFT

3. Calculate the power spectral estimate of the DFT, P;

$$P_i(k) = \frac{1}{N}\left|\sum_{n=1}^{N} s_i(n)h(n)e^{-j2\pi kn/N}\right|^2 \tag{3.3}$$

4. Create a filterbank with M filters linearly spaced in the mel-frequency domain to emulate human auditory perception characteristics;

$$m = 2595 \log_{10}\left(1 + \frac{f}{700}\right) \tag{3.4}$$

$$H_m(k) = \begin{cases} 0 & \text{if } k < f(m-1) \\ \frac{k-f(m-1)}{f(m)-f(m-1)} & \text{if } f(m-1) \leq k \leq f(m) \\ \frac{f(m+1)-k}{f(m+1)-f(m)} & \text{if } f(m) \leq k \leq f(m+1) \\ 0 & \text{if } k > f(m+1) \end{cases} \tag{3.5}$$

where:

m = the Mel-scale obtained from the linear frequency scale f
M = the number of filters in the filterbank
$f(\cdot)$ = the list of $M+2$ Mel-spaced frequencies

5. Filter P with the filterbank and take its logarithm:

$$\text{Spectral Coefficients} = SC_i = P_i(k) \odot H_m(k) \tag{3.6}$$

$$\text{Filterbank Energies} = FE_i = \log(P_i(k) \odot H_m(k)) \tag{3.7}$$

6. Take the DCT of the logarithm of all the filterbank energies to decorrelate and keep the first 13 coefficients:

$$MFCC_i = \sum_{n=0}^{N} FE_{i_n} \cos\left[\frac{\pi}{N}\left(n+\frac{1}{2}\right)k\right] \quad k = 0, \ldots, 12 \tag{3.8}$$

7. Take the first order (deltas) and second order (acceleration) regression coefficients of the MFCC and append them to the the original 13-dimensional vector, obtaining a 39-dimensional vector as the feature vector for the specific frame.

$$\Delta_i = \frac{\sum_{n=1}^{N} n(c_{i+n} - c_{i-n})}{2\sum_{n=1}^{N} n^2} \tag{3.9}$$

$$\Delta\Delta_i = \frac{\sum_{n=1}^{N} n(\Delta_{i+n} - \Delta_{i-n})}{2\sum_{n=1}^{N} n^2} \tag{3.10}$$

where:

Δ_i = the delta coefficient for frame i
$\Delta\Delta_i$ = the acceleration coefficient for frame i
c_i = the static MFCC coefficients of frame i
N = the window length, typically 2

8. The final feature vector $\mathbf{x_i}$ corresponding to the audio frame i is:

$$\mathbf{x_i} = \begin{pmatrix} MFCC_i \\ \Delta MFCC_i \\ \Delta\Delta MFCC_i \end{pmatrix} \tag{3.11}$$

Figure 3.1 shows a speech signal, its filterbank energies and its MFCC.

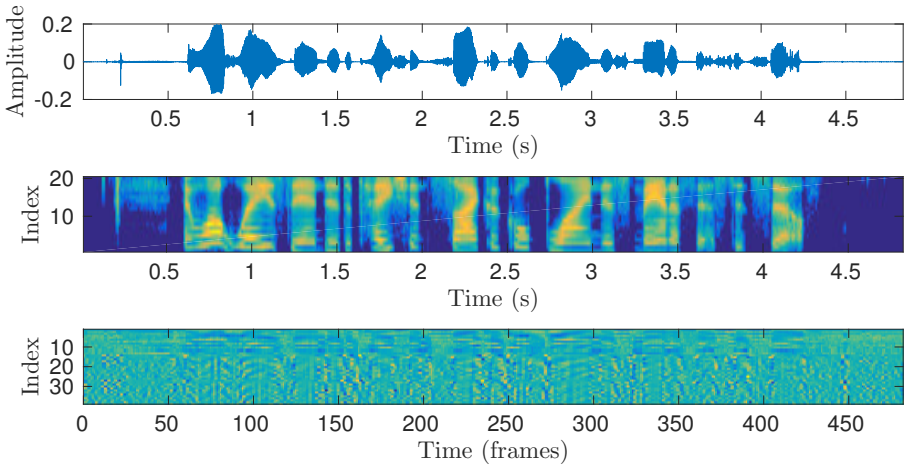


Figure 3.1: A speech signal and its MFCCs. The *temporal envelope* of an utterance (top) is not the ideal candidate for the extraction of linguistic content. The *filter-bank energies* (middle) are better for that purpose, but the *Mel-Frequency Cepstral Coefficients* and their regression coefficients $(\Delta, \Delta\Delta)$ are optimal, since they are decorrelated and they incorporate the change of the feature vector over time *in* the feature vector.

3.3.3 Preprocessing the TIMIT dataset

As stated above, in this work we select 3696, 400 and 192 training, validation and testing utterances respectively. Each utterance is of variable length and therefore, to facilitate efficient processing and storage, all utterances are zero-padded in the time domain in order to have the same length. However, a masking sequence is also created, so that the information of the length of each sequence is not lost.

MFCCs and deltas are then calculated over all utterances and normalized to have zero mean and standard deviation of one. The mean (3.12) and standard deviation (3.13) are estimated over the training set and used to normalize all three subsets according to (3.14).

$$\mu = \frac{1}{N}\sum_{i=1}^{N} X_i \tag{3.12}$$

$$\sigma = \sqrt{\frac{1}{N-1}\sum_{i=1}^{N} |X_i - \mu|^2} \tag{3.13}$$

$$\hat{X} = \frac{X - \mu}{\sigma} \tag{3.14}$$

Using the timestamped, phonetic transcription of each utterance, a label sequence is obtained, with each label associated with the MFCC frame with the same index. Furthermore, the utterances are shuffled within each subset so that the order of

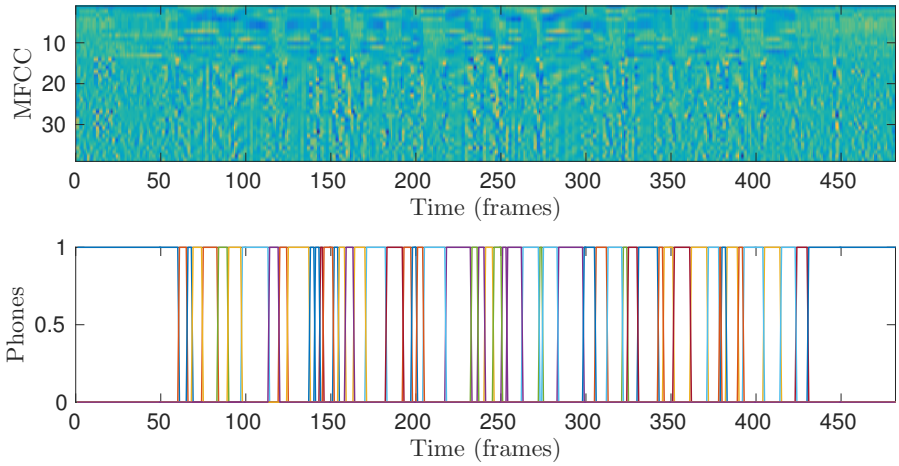


Figure 3.2: A preprocessed TIMIT sentence. After preprocessing, a TIMIT sentence consists of its MFCCs and deltas sequence (top) and a sequence of target phone labels that correspond to the frames of the MFCC sequence (bottom). Each colour represents a different phone.

appearance doesn't correlate with the original order of the original utterances (dialect/speaker/sentence type).

Figure 3.2 shows a feature vector sequence of MFCCs and its target label sequence that correspond to one TIMIT utterance.

3.3.4 Storing the preprocessed dataset

After preprocessing the audio signals, the 39-dimensional feature vector sequences, the masking sequences and the label sequences are stored, separately for all three subsets, into a single file using the widely supported hierarchical HDF5 format, ensuring compatibility and ease of access by all modern frameworks and programming languages.

3.4 Conclusion

The TIMIT dataset, a speech corpus containing spoken English sentences and annotated at the phone level, was preprocessed to facilitate the efficient training of LSTM Automatic Speech Recognition model. After its processing, the bundle of necessary data contains sequences of cepstral coefficients that correspond to the audio signals, sequences of target phone labels of equal length, and masking sequences that preserve the information about each sequence's original length. The data is normalized, shuffled and partitioned to a training set (3696 sequences), a validation set (400 sequences) and a test set (192 sequences).

In the following chapter, the LSTM architecture that is trained on the TIMIT data is presented.

Chapter 4

An LSTM Speech Recognition System

In this chapter, an Automatic Speech Recognition system based on LSTM is presented and implemented in the Torch Framework[4][5]. The chapter begins by motivating the selection of an LSTM-based architecture and proceeds to describe the architecture, the testing and evaluation process and finally the trained parameters of the system.

4.1 Motivation for Long Short-Term Memory

4.1.1 End-to-end Machine Learning

Neural network architectures do not rely, in general, on hand-crafted and domain-specific feature extraction. Instead, they learn these features at different levels of abstraction, this being one of the major advantages of deep learning algorithms. This is in accord with the general *end-to-end trend* that has emerged in machine learning, i.e. the minimization of expert knowledge and preprocessing [12]. Therefore, the research can focus on the tuning of the network hyperparameters that yield best performance, on the optimization of the training procedure and, in the case of this work, on the optimization of the inference procedure (the forward pass) in terms of precision and power consumption.

4.1.2 Native temporal context incorporation

LSTM architectures incorporate long and short temporal dependencies natively and scale well with the time lag between important events [13]. Their size and complexity do not explode exponentially with the amount of available context as they would on a fully connected neural network with buffered inputs. This allows for small networks (comparatively with other architectures), which in turn means shorter training and inference times, and, with respect to hardware, significantly smaller power consumption and fewer calculations.

4.1.3 Homogeneous computations

From a computational complexity point of view, the functionality of neural networks can be thought as sets of matrix-vector multiplications, element-wise vector additions and products and element-wise application of a limited number of activation functions, such as the logistic sigmoid or the hyperbolic tangent. This homogeneity can be a key point in the design of optimized, power-efficient hardware for ASR and machine learning in general, since the designers can focus on optimizing these and only these time and energy consuming operations.

4.2 The LSTM

LSTM blocks are in the heart of this ASR Model. A schematic of the LSTM architecture can be seen in Figure 4.1. LSTM blocks feature memory cells which can maintain their state over time and normal and fully connected non linear units that regulate how information flows into and through the block.

The LSTM architecture features an internal *cell state*, which can be thought of as the memory of the LSTM. Furthermore, it has four "gates", the *forget, input, output* and *candidate state* gates. Each of them is a small RNN with dimensions $d \times h$ where d is the dimension of the input vector $\mathbf{x_t}$ and h is the dimension of the cell state $\mathbf{C_t}$ and LSTM output $\mathbf{h_t}$, also called the *hidden size*. Each of these gates interact with the cell state, controlling how new information entering the LSTM will alter it. The equations that describe the forward pass of an LSTM block are stated and explained below.

The *forget gate* controls how much the current input will influence *the removal of information* from the previous cell state.

$$\mathbf{f_t} = \sigma(\mathbf{W_f} \cdot \mathbf{x_t} + \mathbf{R_f} \cdot \mathbf{h_{t-1}} + \mathbf{b_f}) \tag{4.1}$$

The *input gate* controls how much the current input will influence *the addition of new information* to the current cell state.

$$\mathbf{i_t} = \sigma(\mathbf{W_i} \cdot \mathbf{x_t} + \mathbf{R_i} \cdot \mathbf{h_{t-1}} + \mathbf{b_i}) \tag{4.2}$$

The *output gate* controls how much the current input will *directly influence the current output* of the network.

$$\mathbf{o_t} = \sigma(\mathbf{W_o} \cdot \mathbf{x_t} + \mathbf{R_o} \cdot \mathbf{h_{t-1}} + \mathbf{b_o}) \tag{4.3}$$

The *candidate stage gate* is *the representation of new information* created by the current input.

$$\tilde{\mathbf{C}}_\mathbf{t} = \tanh(\mathbf{W_C} \cdot \mathbf{x_t} + \mathbf{R_C} \cdot \mathbf{h_{t-1}} + \mathbf{b_C}) \tag{4.4}$$

The *current cell state* is a combination of *things to be forgotten* and *things to be assimilated*.

$$\mathbf{C_t} = \mathbf{f_t} \odot \mathbf{C_{t-1}} + \mathbf{i_t} \odot \tilde{\mathbf{C}}_\mathbf{t} \tag{4.5}$$

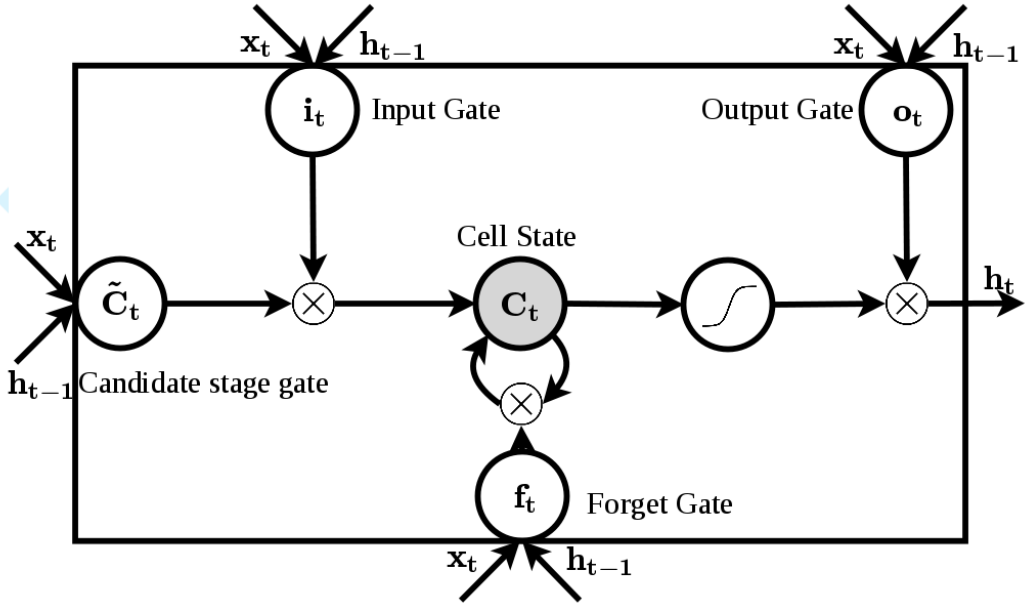


Figure 4.1: The LSTM architecture. It consists of an internal *cell* that has a *state*. The MFCC inputs $\mathbf{x_t}$ are combined with the previous outputs $\mathbf{h_{t-1}}$ to alter the cell state $\mathbf{C_{t-1}}$ and to output a new prediction, $\mathbf{h_t}$. Context from all previous inputs is incorporated into the new cell state $\mathbf{C_t}$, and therefore, the LSTM can take into account temporal dependencies that are very far apart.

The *network output* is a combination of the input and the current cell state.

$$\mathbf{h_t} = \mathbf{o_t} \odot \tanh(\mathbf{C_t}) \tag{4.6}$$

where:

$\mathbf{x_t}$	= the current input vector
$\mathbf{h_t}$	= the current LSTM output
$\mathbf{h_{t-1}}$	= the previous LSTM output
$\mathbf{f_t}, \mathbf{i_t}, \mathbf{o_t}$	= the output of the forget, input and output gates
$\tilde{\mathbf{C}}_\mathbf{t}$	= the output of the candidate state gate
$\mathbf{C_t}$	= the current cell state
$\mathbf{C_{t-1}}$	= the previous cell state
$\mathbf{W_f}, \mathbf{W_i}, \mathbf{W_o}, \mathbf{W_C}$	= the input weights for gates $\mathbf{f_t}$, $\mathbf{i_t}$, $\mathbf{o_t}$ and $\tilde{\mathbf{C}}_\mathbf{t}$
$\mathbf{R_f}, \mathbf{R_i}, \mathbf{R_o}, \mathbf{R_C}$	= the recurrent weights for gates $\mathbf{f_t}$, $\mathbf{i_t}$, $\mathbf{o_t}$ and $\tilde{\mathbf{C}}_\mathbf{t}$
$\mathbf{b_f}, \mathbf{b_i}, \mathbf{b_o}, \mathbf{b_C}$	= the bias weights for gates $\mathbf{f_t}$, $\mathbf{i_t}$, $\mathbf{o_t}$ and $\tilde{\mathbf{C}}_\mathbf{t}$

Using these mechanisms, during training the LSTM can learn to discriminate between relevant and irrelevant context, by adjusting the weights of its gates. During the inference process, the output of the LSTM block $\mathbf{h_t}$ is a combination of both the current input $\mathbf{x_t}$ and the cell state $\mathbf{C_t}$, as it has been "sculpted" from the previous inputs, via their passage through the four gates.

To sum up, given an input sequence $\mathbf{X} = (\mathbf{x_1}, \ldots, \mathbf{x_T})$ an LSTM Neural Network computes its output sequence $\mathbf{H} = (\mathbf{h_1}, \ldots, \mathbf{h_T})$ by evaluating equations 4.1 to 4.6. The procedure is demonstrated in pseudocode in Algorithm 1.

Algorithm 1 LSTM Forward Pass

procedure ForwardLSTM($\mathbf{X}$) ▷ Forward pass for a sequence $\mathbf{X}$ of input vectors $\mathbf{x_t}$
  $\mathbf{C_0} \leftarrow \mathbf{0}$
  $\mathbf{h_0} \leftarrow \mathbf{0}$
  for every sequence frame $\mathbf{x_t}$, $1 \leq t \leq T$ **do** ▷ T is the sequence length
    $\mathbf{f_t} = \sigma(\mathbf{W_f} \cdot \mathbf{x_t} + \mathbf{R_f} \cdot \mathbf{h_{t-1}} + \mathbf{b_f})$ ▷ The forget gate
    $\mathbf{i_t} = \sigma(\mathbf{W_i} \cdot \mathbf{x_t} + \mathbf{R_i} \cdot \mathbf{h_{t-1}} + \mathbf{b_i})$ ▷ The input gate
    $\mathbf{o_t} = \sigma(\mathbf{W_o} \cdot \mathbf{x_t} + \mathbf{R_o} \cdot \mathbf{h_{t-1}} + \mathbf{b_o})$ ▷ The output gate
    $\tilde{\mathbf{C}}_\mathbf{t} = \tanh(\mathbf{W_C} \cdot \mathbf{x_t} + \mathbf{R_C} \cdot \mathbf{h_{t-1}} + \mathbf{b_C})$ ▷ The candidate state
    $\mathbf{C_t} = \mathbf{f_t} \odot \mathbf{C_{t-1}} + \mathbf{i_t} \odot \tilde{\mathbf{C}}_\mathbf{t}$ ▷ The current state
    $\mathbf{h_t} = \mathbf{o_t} \odot \tanh(\mathbf{C_t})$ ▷ The current output
  end for
end procedure ▷ Output is stored in the sequence $\mathbf{H}$ of vectors $\mathbf{h_t}$

4.3 The Bidirectional LSTM

In this work, two identically sized LSTM blocks are used in parallel, one processing the input sequence in a forward manner (from the 1st to the T-th frame) and the other processing the input sequences in a backward manner (from the T-th to the 1st frame). This configuration, called Bidirectional LSTM or BLSTM, has proven to drastically increase the performance of LSTM-based phoneme recognizers [14] [35]. This process requires the "existence" of the whole utterance *before* phone recognition can begin, so it seems to rely on "future knowledge". However, this is not actually the case, since speech recognition is not a truly online task (i.e. it does not require an output after every input). An output at the end of every speech segment or sentence is sufficient. This technique also mimics how we humans process chunks of heard speech; When unsure for a spoken word or sound, we use both previous and subsequent context to reach a confident decision.

4.4 The Fully Connected Neural Layer

Since the hidden size (i.e. the dimensionality of $\mathbf{h_t}$ and also $\mathbf{C_t}$) of the LSTMs is a tunable hyperparameter that has shown to influence the amount of context that the model can take into account and also its overall performance, a fully connected layer is used as an output layer, classifying the patterns that the LSTM blocks as one of 61 phone labels, while allowing for easy tuning of the hyperparameter h.

The equation that describes the forward pass of the fully connected neural network is shown below. Since the fully connected layer is used as an output, no activation

function is used.

$$\mathbf{l_t} = \mathbf{W_{FC}} \cdot \{\mathbf{Fh_t}, \mathbf{Bh_t}\} + \mathbf{b_{FC}} \tag{4.7}$$

where:

$\mathbf{l_t}$	= the output vector of target probabilities
$\{\mathbf{Fh_t}, \mathbf{Bh_t}\}$	= the concatenated outputs of the two LSTM blocks
$\mathbf{W_{FC}}$	= the $2h \times l$ weight matrix of the layer
$\mathbf{b_{FC}}$	= the l-dimensional bias vector of the layer

The forward pass of a fully connected neural network is described in pseudocode in Algorithm 2.

Algorithm 2 Fully Connected Neural Network Forward Pass

```
1:  procedure ForwardFC(x)                    ▷ Forward pass for an input vector x
2:      for 1 ≤ i ≤ M do                      ▷ M is the output vector size
3:          z_i ← 0
4:          for 1 ≤ j ≤ N do                  ▷ N is the input vector size
5:              z_i ← z_i + w_ji x_j          ▷ w is the weight matrix
6:          end for
7:          z_i ← z_i + b_i                   ▷ b is the bias vector
8:          y_i ← g(z_i)                      ▷ g(·) is the activation function
9:      end for
10: end procedure                             ▷ Output is stored in vector y
```

4.5 Overall Architecture

In this work, a LSTM / Fully Connected Neural Network architecture was implemented, trained and benchmarked on the TIMIT Speech Corpus.

The high level architecture can be seen in the Figure 4.2. The first layer of the network consists of an LSTM network with two parallel LSTM blocks (a Bidirectional LSTM). The second layer consists of a fully connected layer that connects the outputs of both the parallel blocks to the output.

The procedure of evaluating the outputs of the LSTM/FC architecture is demonstrated in pseudocode in Algorithm 3.

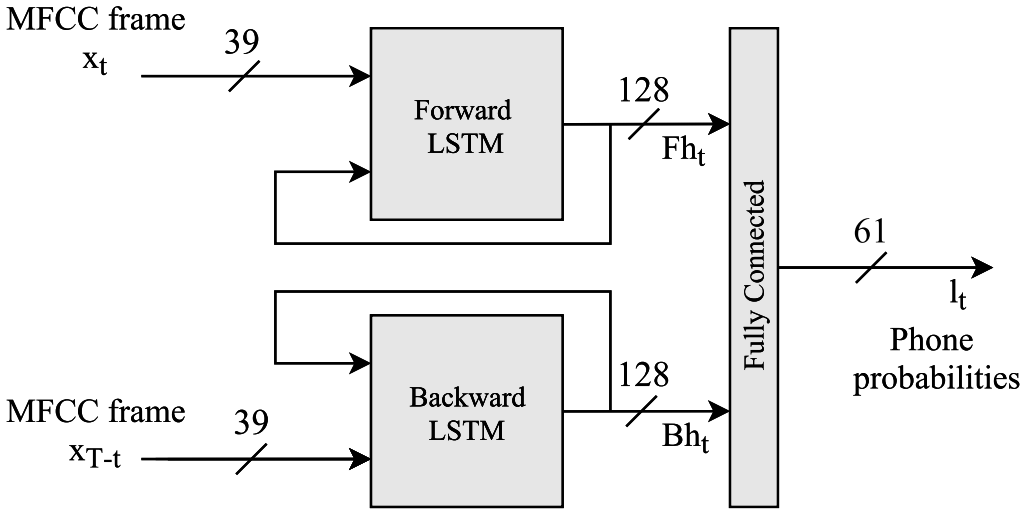


Figure 4.2: The overall LSTM/FC architecture. A preprocessed MFCC sequence $\mathbf{X} = (\mathbf{x_1}, \ldots, \mathbf{x_T})$ is passed on to one LSTM block, frame by frame. The sequence $\mathbf{X}$ is also inverted to create sequence $\mathbf{Y} = (\mathbf{y_1}, \ldots, \mathbf{y_T}) = (\mathbf{x_T}, \ldots, \mathbf{x_1})$. The sequence $\mathbf{Y}$ is passed on to the other LSTM block. Patterns recognized from the two LSTM blocks are concatenated and passed to the fully connected layer that classifies them as 1-of-61 phones. In this way, all temporal context from one MFCC sequence is used in the prediction of the phones.

Algorithm 3 LSTM/FC Forward Pass

1: **procedure** ForwardLSTM/FC($\mathbf{X}$)▷ Forward pass for a sequence $\mathbf{X}$ of input vectors $\mathbf{x_t}$
2: ForwardLSTM($\mathbf{X}$) for the forward LSTM layer storing outputs $\mathbf{Fh_t}$ in sequence $\mathbf{FH}$
3: Flip sequence $\mathbf{X}$ to create $\mathbf{Y}$
4: ForwardLSTM($\mathbf{Y}$) for the backward LSTM layer storing $\mathbf{Bh_t}$ in sequence $\mathbf{BH}$
5: **for all** t, in any order **do**
6: ForwardFC($\mathbf{Fh_t}, \mathbf{Bh_t}$) for the Fully Connected Layer
7: **end for**
8: **end procedure**

4.6 Software LSTM/FC Implementation in Torch

The aforementioned LSTM/FC machine learning architecture is implemented in the Torch machine learning framework [4]. Since the data was preprocessed to 39-dimensional MFCC coefficients, the input layer was of size $d = 39$. The hidden layer of each of the parallel blocks had a size of $h = 128$, therefore the fully connected layer had an input size of $2h = 256$. The output layer had a size of $l = 61$, since the complete set of 61 phones provided in the transcriptions was used. The LSTM blocks use the logistic sigmoid $\sigma(x) = \frac{1}{1+e^{-x}}$ activation function for gates $\mathbf{f}, \mathbf{i}$ and $\mathbf{o}$ and the hyperbolic tangent $\tanh(x)$ activation function for the candidate stage gate $\tilde{\mathbf{C}}$ and for the computation of the output $\mathbf{h_t}$. The fully connected layer uses the Logarithmic Softmax function on its output, but only during training.

Figure 4.3 shows the full functionality of the developed Automatic Speech Recognition System. For each MFCC frame in a sequence, representing 25ms of speech, the developed architecture takes into account the full phonetic context and outputs a probability vector of size $l = 61$, with each element representing the probability of each one of the 61 phones having occurred in the frame. Each coloured line corresponds to the probability of one phone.

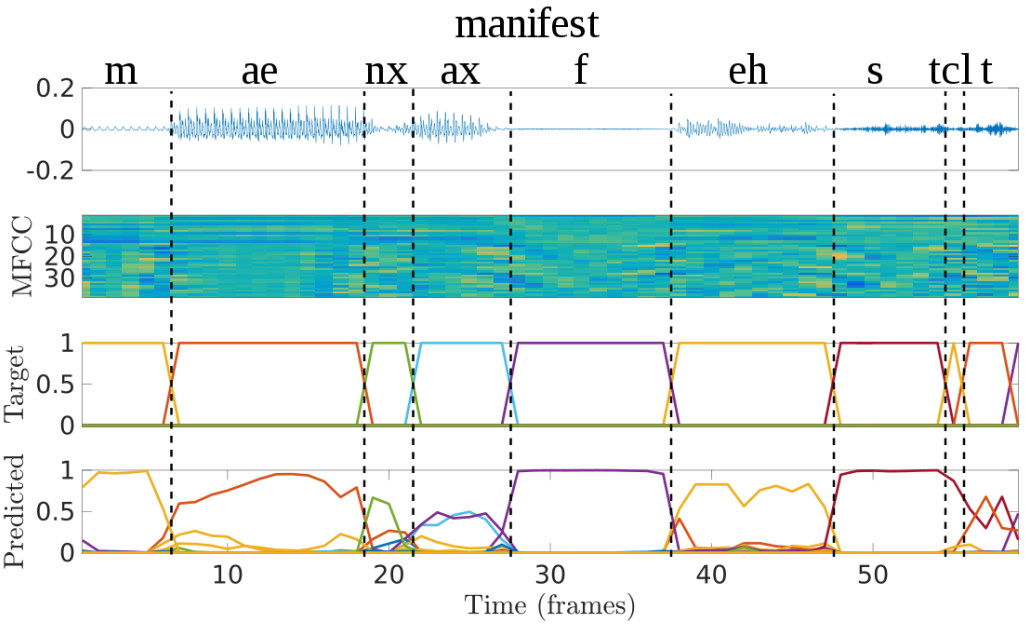


Figure 4.3: The functionality of the LSTM/FC architecture. Top: Speech signal of the word *manifest*. Second from top: The MFCC sequence $\mathbf{X} = (\mathbf{x_1}, \ldots, \mathbf{x_T})$ that corresponds to the utterance. Second from bottom: The correct label sequence. Bottom: The network output $\mathbf{L} = (\mathbf{l_1}, \ldots, \mathbf{l_T})$, where each vector $\mathbf{l_t}$ represents the predicted probability of each of 1-of-61 phones. Each coloured line indicates the probability of each phone.

4.6.1 Training

The architecture was trained on TIMIT data using the full Backpropagation Through Time (BPTT) algorithm[34]. All the network parameters were initialized from a uniform distribution in $[-0.1, 0.1]$ and the learning rate was initialized to $5 * 10^3$.

Each utterance, i.e. sequence of frames, was fed to both LSTM parallel blocks, in forward order to the one and in backward order to the other. The output of both these blocks was concatenated as a vector $\{\mathbf{Fh_t}, \mathbf{Bh_t}\}$ of size $2h$ and fed to the fully connected layer. A softmax activation function was at the output of the fully connected layer, and the cross entropy error function was used as a loss criterion, as is standard for 1 of N classification. At the end of each sequence, weight updates were carried out according to the BPTT method and the network's activations were reset to 0. The full Torch code that implements this procedure can be found in Appendix A.1.

The cross entropy loss function is defined as:

$$\text{Loss}(\mathbf{l}, class) = -l_{class} + \log\left(\sum_{k=1}^{K} e^{l_k}\right) \tag{4.8}$$

where:

$\mathbf{l}$ = the output log-probability vector
$class$ = the target class
K = the output vector dimension i.e. the number of possible classes

Because the architecture processes sequences of inputs, the total sequence cross entropy loss was taken into account, and is defined below:

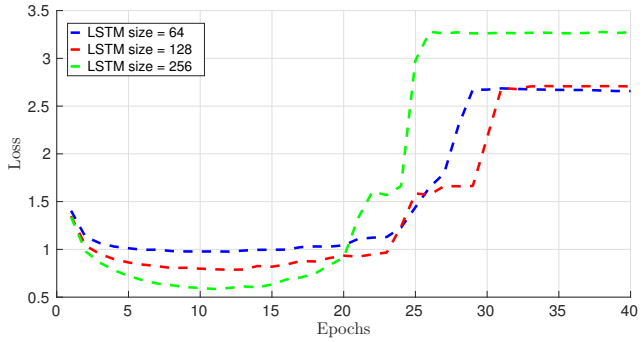
$$\text{Sequence Loss} = \sum_{i=1}^{T} \text{Loss}(\mathbf{l_i}, class_i) \tag{4.9}$$

where:

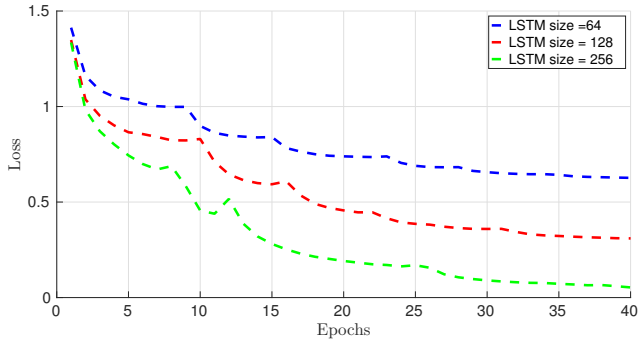
$\mathbf{l_i}$ = the output vector for frame i
$class_i$ = the target class for frame i
T = the sequence length in frames

By incorporating the softmax layer at the output, the 61 outputs of this architecture can be interpreted as the posterior probabilities of each one of the 61 phones existing in the current frame, given all the inputs in the current sequence. This happens because the Bidirectional LSTM incorporates all utterance context, both previous and subsequent.

The architecture was trained for 40 epochs, but its parameters were stored in "snapshots" every epoch. After every epoch, the performance of the network was tested on the validation set. Two different training modes were implemented: One with a non-adaptive learning rate and one that implemented an adaptive, "bold driver" strategy for the changing of learning rate. This strategy slowly increases the



(a) Non-Adaptive Learning Rate



(b) Adaptive Learning Rate

Figure 4.4: Loss function vs. Epochs. (a) When the learning rate is fixed at $5 * 10^{-3}$, the network fails to converge. (b) By adaptively changing the learning rate, the loss function over the training set is minimized as epochs advance. Loss curves for LSTM sizes of 64(blue), 128(red) and 256(green) are shown.

learning rate if the loss function is decreasing and sharply decreases the learning rate when the loss function increases. This implementation can be found in appendix A.1.2.

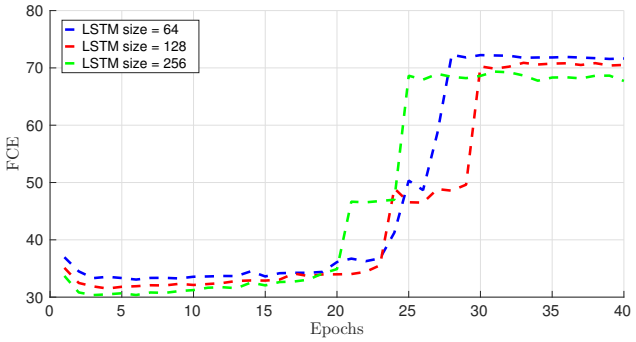
Figure 4.4 shows the cross-entropy loss over the epochs of training. Both learning rate approaches can be seen. In both cases, the network demonstrates remarkably fast learning compared to other non-LSTM architectures[14], but in the case of non-adaptive learning rate, the network amusingly *forgets everything it has learned* and fails to converge.

4.6.2 Testing

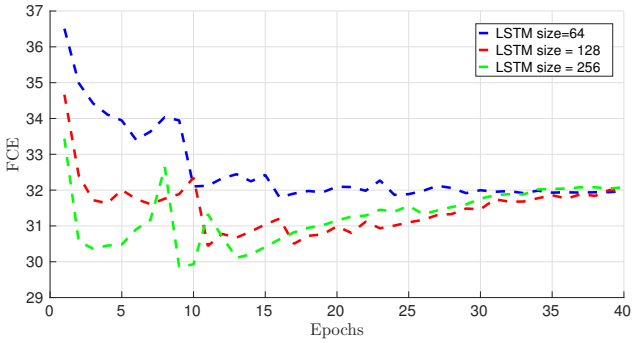
The metric used to quantify the performance of this architecture is the Framewise Phone Classification Error:

$$\text{FCE} = 1 - \text{Accuracy} = \frac{I}{T} \tag{4.10}$$

where:



(a) Non-Adaptive Learning Rate



(b) Adaptive Learning Rate

Figure 4.5: Framewise Classification Error vs. Epoch. When the learning rate is fixed at $5 * 10^{-3}$, the network achieves a minimum FCE in 4-5 epochs, but then diverges. When using an adaptive learning rate, even though the loss continues to decrease as training continues, the FCE over the validation set is minimized after a specific amount of epochs (around 10) and then begins to slowly rise. This suggests overfitting of the training data and indicates a possibility for even better performance if the network is trained with more data. Curves for LSTM sizes of 64 (blue), 128 (red) and 256(green) are shown.

T = the sequence length in frames
I = the number of incorrectly classified frames.

The Framewise Classification Error (FCE) denotes the amount of the frames in a sequence that got classified wrongly with respect to the total number of frames in the sequence. The FCE was averaged over the validation set after each epoch. The process of evaluating and scoring can be found in appendix A.1.3. The average validation set FCE over epochs can be shown in Figure 4.5.

The rationale behind the selection of this metric lies in the fact that lowering this error metric requires effective usage of contextual information. For each input sequence $\mathbf{X}$ of length T representing a TIMIT spoken utterance, a corresponding sequence $\mathbf{l}$ is output from the network, as explained in 4.2. Since each output $\mathbf{l_t}$ represents the probabilities of the 61 phones existing in a frame, the phone with

the largest probability is regarded as the final label produced by the network. This predicted label is compared to the actual label of the frame, and if they match, then the network has made a correct prediction (a hit) and if it does not, the network has made an incorrect prediction (a miss). The scoring method is outlined in pseudocode in Algorithm 4.

Algorithm 4 Performance Scoring

```
Average FCE ← 0
for i=1 to dsSize do                    ▷ dsSize the number of sequences in the set
    for j=1 to T do                                  ▷ T the length of the sequence
        I ← 0
        if predicted label ≠ target label then
            I ← I + 1                ▷ I the number of incorrenctly classified frames
        end if
    end for
    FCE_i ← I/T
    Average FCE ← Average FCE + FCE_i
    Average FCE ← Average FCE/dsSize
end for
```

Even with adaptive learning rate, the network's performance continues to increase over the training set, but starts decreasing over the validation set after achieving peak performance in about 10 epochs. This behaviour suggests overfitting of the training data, and it might be caused by the network's better adaptation to long range regularities such as phone ordering within words or speaker specific pronunciations [14]. If this suggestion is true, then it might be worth exploring the behaviour of the LSTM/FC architecture on tasks with more training data.

The best performing network has a size of 256, is trained with adaptive learning rate and shows a 29.8% FCE, which corresponds to 70.2% accuracy after being trained for 10 epochs. However, for the rest of this work, we select a network with size 128 which achieves an 30.2% FCE, since it has roughly half the parameters, allowing for faster execution of experiments, while not hurting generality. It should be also noted that this architecture can be implemented in a bigger ASR chain, where other recognition components can be used. This includes language-aware systems that can make statistical assumptions about the neural architectures output, effectively correcting its predictions via language/grammar/vocabulary-aware methods.

4.6.3 The architecture's learned parameters

The LSTM/FC architecture described above has a wide collection of trainable parameters. These parameters are better represented as weight matrices and bias vectors: The element w_{ij} of a weight matrix $\mathbf{W}$ is the weight of the connection between the input i and the output j. Accordingly, an element b_j of a bias vector $\mathbf{b}$ is the bias added to the output j.

There are three major sets of parameters: Those that belong to the Forward LSTM, denoted with the prefix **F**, those that belong to the Backward LSTM, denoted with the prefix **B** and those that belong to the fully connected output layer, denoted with the prefix **FC**.

Another major classification of the parameters is whether they are feed-forward connections, denoted with the letter **W** involving calculations with current input $\mathbf{x_t}$, recurrent connections, denoted with the letter **R** involving calculations with previous output $\mathbf{h_{t-1}}$, or biases, denoted with the letter **b**.

The final classification of the parameters has to do with the gate that they are associated with: The index can be one of $\{\mathbf{f}, \mathbf{i}, \mathbf{o}, \mathbf{C}\}$ and correspond to the forget, input, output and candidate cell state gates.

As an example, $\mathbf{FR_i}$ represents the weights of the recurrent input gate connections of the forward LSTM, while $\mathbf{BF_o}$ represents the weights of the feed-forward output gate connections that belong to the Backward LSTM.

During training, all these parameter matrices are constantly updated, with the ultimate goal of convergence to a set of values that will maximize the architecture's predictive performance. In other words, neural connections get reinforced or weakened based on an iterative procedure. However, the mathematical and algorithmic nature of this update procedure is not easy to understand formally.

For that reason, after the training and the selection of the best performing snapshot of the model, the learned parameters were examined from a statistical point of view. This step was implemented because it exposes approximate computing opportunities that otherwise would go unnoticed.

Figure 4.6 shows boxplot and histogram figures of the matrices of learneable parameters of every layer in the architecture, while Table 4.1 summarizes the statistical properties of the parameters of each layer. The properties that are reported are the *mean, standard deviation, minimum, maximum, median*, and the 1st and 3rd *quartiles*. Definitions of the mean and the standard deviation were given in equations 3.12 and 3.13. For a random variable vector X of length K, when ordered in ascending order of magnitude, the *inter-quartile range* and the lower and upper fences of its elements are defined as follows:

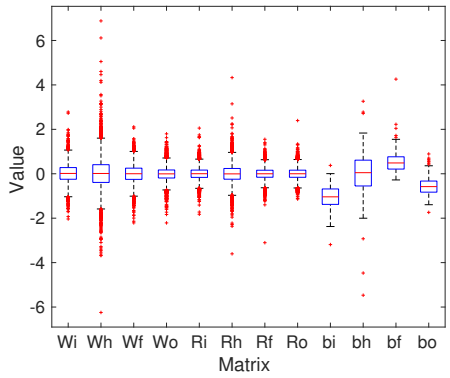
$$\text{Inter-quartile Range} = IQR = Q_3 - Q_1 \tag{4.11}$$

$$\text{Lower Fence} = LF = Q_1 - 1.5(IQR) \tag{4.12}$$

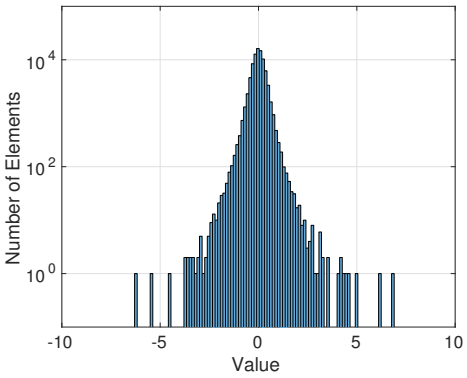
$$\text{Upper Fence} = UF = Q3 + 1.5(IQR) \tag{4.13}$$

where:

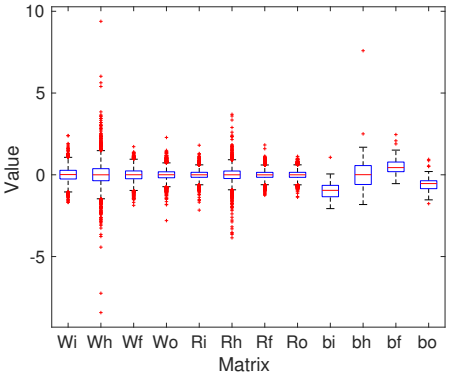
$Q_1 = \frac{1}{4}(K+1)$th element of vector X
$Q_2 = \frac{1}{2}(K+1)$th element of vector X i.e the *median*
$Q_3 = \frac{3}{4}(K+1)$th element of vector X.



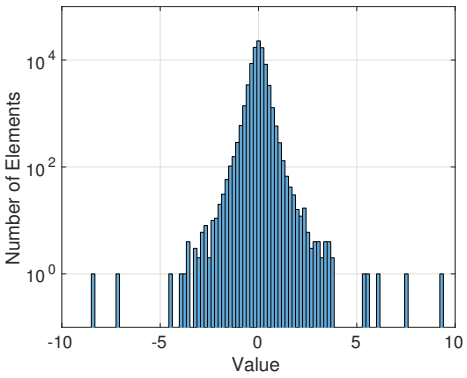
(a) Forward LSTM boxplot



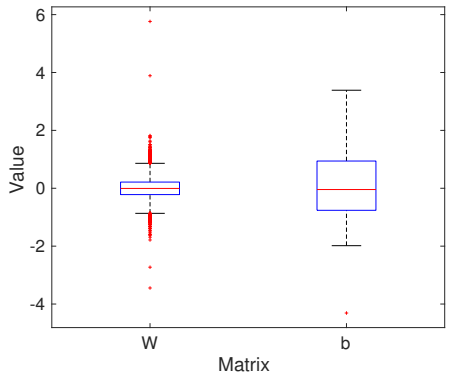
(b) Forward LSTM aggregate distribution



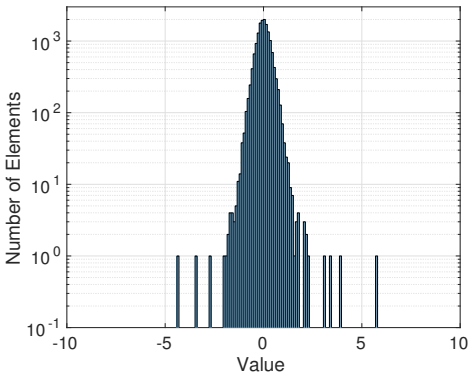
(c) Backward LSTM boxplot



(d) Backward LSTM aggregate distribution



(e) Fully Connected boxplot



(f) Fully Connected aggregate distribution

Figure 4.6: Architecture Parameters. Both per-layer boxplots and aggregate distribution log-histograms are presented for the Forward LSTM (top), Backward LSTM (middle), and Fully Connected layer (bottom). Outliers extend up to 17σ for some layers. Note the log-axis in the histograms.

Layer	Elements	μ	σ	min	Q_1	median	Q_3	max
Forward LSTM								
FWi	4,992	$1.49 \cdot 10^{-2}$	0.44	−2.04	−0.25	$1.45 \cdot 10^{-2}$	0.28	2.78
FWc	4,992	$8 \cdot 10^{-3}$	0.77	−6.24	−0.39	$1.29 \cdot 10^{-2}$	0.41	6.88
FWf	4,992	$-4.6 \cdot 10^{-3}$	0.42	−2.21	−0.25	$1.25 \cdot 10^{-3}$	0.25	2.11
FWo	4,992	$-7.8 \cdot 10^{-3}$	0.34	−2.21	−0.19	$-4.33 \cdot 10^{-3}$	0.17	1.8
FRi	16,384	$2.1 \cdot 10^{-3}$	0.26	−1.82	−0.16	$2.14 \cdot 10^{-3}$	0.17	2.06
FRc	16,384	$-4.3 \cdot 10^{-3}$	0.39	−3.6	−0.24	$-5.08 \cdot 10^{-3}$	0.24	4.33
FRf	16,384	$9.89 \cdot 10^{-4}$	0.25	−3.1	−0.16	$-5.95 \cdot 10^{-4}$	0.16	1.55
FRo	16,384	$1.5 \cdot 10^{-3}$	0.25	−1.14	−0.16	$2.54 \cdot 10^{-4}$	0.16	2.39
Fbi	128	−1.06	0.55	−3.19	−1.38	−1.04	−0.68	0.38
Fbc	128	$1.47 \cdot 10^{-2}$	1.13	−5.47	−0.55	$4.87 \cdot 10^{-2}$	0.61	3.26
Fbf	128	0.57	0.57	−0.28	0.21	0.49	0.76	4.26
Fbo	128	−0.54	0.46	−1.74	−0.83	−0.58	−0.33	0.89
Backward LSTM								
BWi	4,992	$8.1 \cdot 10^{-3}$	0.42	−1.69	−0.26	$1.39 \cdot 10^{-2}$	0.27	2.4
BWc	4,992	$8.8 \cdot 10^{-3}$	0.74	−8.43	−0.36	$-5.21 \cdot 10^{-4}$	0.37	9.38
BWf	4,992	$-5.1 \cdot 10^{-3}$	0.39	−1.87	−0.24	$3.39 \cdot 10^{-3}$	0.24	1.72
BWo	4,992	$-4.56 \cdot 10^{-4}$	0.33	−2.8	−0.18	$2.78 \cdot 10^{-3}$	0.18	2.28
BRi	16,384	$-1.3 \cdot 10^{-3}$	0.25	−2.16	−0.15	$5.05 \cdot 10^{-5}$	0.15	1.81
BRc	16,384	$-1.21 \cdot 10^{-4}$	0.38	−3.85	−0.23	$-3.06 \cdot 10^{-4}$	0.23	3.7
BRf	16,384	$-2.4 \cdot 10^{-3}$	0.24	−1.26	−0.15	$-2.34 \cdot 10^{-3}$	0.15	1.82
BRo	16,384	$-1.2 \cdot 10^{-3}$	0.24	−1.37	−0.16	$-1.59 \cdot 10^{-4}$	0.15	1.13
Bbi	128	−0.97	0.48	−2.07	−1.34	−0.95	−0.65	1.07
Bbc	128	$7.29 \cdot 10^{-2}$	1.08	−1.82	−0.59	$7.36 \cdot 10^{-3}$	0.56	7.59
Bbf	128	0.5	0.47	−0.54	0.19	0.44	0.78	2.46
Bbo	128	−0.57	0.42	−1.77	−0.85	−0.54	−0.36	0.94
Fully Connected								
FCw	15,616	$-4.67 \cdot 10^{-4}$	0.37	−3.44	−0.22	$-5.69 \cdot 10^{-3}$	0.21	5.76
FCb	61	0.13	1.4	−4.31	−0.76	$-4.37 \cdot 10^{-2}$	0.94	3.39

Table 4.1: Statistical properties of the architecture's parameters. The large number of outliers suggests that the network is *heavily influenced* from a small number of connections. This observation motivates *connection pruning* and will be investigated in Chapter 5.

4.7 Conclusion

Both from a machine learning and a computational/hardware perspective, there is much motivation in utilizing Recurrent Neural Network architectures to process sequential tasks. More specifically, the ability of the LSTM architecture to model long-term context while keeping its size manageable and its internal calculations homogeneous, make it an ideal candidate for hardware-optimized machine learning solutions that solve tasks of sequential nature.

In this work, an architecture that incorporates two LSTM blocks and one fully connected layer, called LSTM/FC, was implemented in the Torch machine learning framework and trained over the TIMIT speech corpus, which consists of spoken English sentences annotated to the phone level. The Framewise Phone Classification Error (FCE) was used as a metric of performance, and the best performing network

achieved an FCE of 29.8%. This corresponds to 70.2% accuracy in phone prediction. The architecture showed remarkable ability to learn, achieving peak performance significantly close to the state of the art in much less time than other, non-recurrent architectures.

In the following chapter, the mathematical inner workings of this architecture will be exposed and analyses will be performed to identify optimization opportunities with respect to a hardware implementation. The natural error resilience of neural networks will be exploited through approximate computing to yield an efficient hardware model with no significant loss in prediction performance but with significantly reduced energy footprint.

Chapter 5

Approximate computing on LSTMs

5.1 Introduction

5.1.1 The error resilience of Neural Networks

As early as 1956, John von Neumann in his work "Probabilistic Logics and the Synthesis of Reliable Organisms from Unreliable Components"[32] addressed the fact that systems with large numbers of functional units and interconnections can intrinsically create a distributed representation of the problems they model. At the same time and in a different discipline, it is shown that biological neural networks of developed animals are tolerant against damage to individual neurons or synapses. In the field of Electrical Engineering and Mathematics, this natural error resilience of neural architectures has been since well-documented, but its applications had not been thoroughly exploited until very recently.

5.1.2 Approximate Computing

Approximate computing is the computational and design paradigm in which a designer is willing to sacrifice a tolerable amount of accuracy of his application in favour of significant resource savings. Resource savings can take many forms, depending on the field. In hardware and electronics design, these are most commonly energy consumption, silicon area and critical path length. In this case, it is noteworthy that allowing for bounded approximation is limited to the datapath and not to control structures, since control operations and circuitry are fault-critical and even the slightest miscalculation could potentially lead to aberrant system behaviour.

In this chapter, specific approaches of approximate computing will be taken in order to simplify, in a hardware, software, algorithmic and mathematical level the architecture described in Chapter 4. After each approach, an assessment on performance degradation will be conducted, the most favourable tradeoffs will be identified and a proposed exploit method will be outlined.

Operation	Energy [pJ]	Relative Cost
32 bit int ADD	0.1	1
32 bit float ADD	0.9	9
32 bit Register File	1	10
32 bit int MULT	3.1	31
32 bit float MULT	3.7	37
32 bit SRAM Cache	5	50
32 bit DRAM Memory	640	6,400

Table 5.1: Energy consumption for basic arithmetic and memory operations in a 45nm CMOS process.

5.2 Motivation for approximate NNs

Nowadays, applying approximate computing to neural network architectures is increasingly regarded as a very promising approach to leveraging their aforementioned error resilience, significantly reducing computational cost and energy consumption with imperceptible or at least small sacrifices in prediction performance.

Depending on the task, Neural Network architectures can have thousands, millions or even hundred millions of learnable parameters. This makes them both computationally intensive and memory intensive, which is a definite disadvantage when it comes to deploying them on low-power autonomous embedded systems. Table 5.1 shows typical energy values for basic arithmetic and memory operations for a 45nm CMOS process[18].

In this work, three basic approaches of approximate computing are applied on the LSTM/FC network implemented in Chapter 4:

1. *Quantization of the network's parameters* to ever-decreasing word lengths;
2. *Connection pruning* of an increasing amount of neural connections;
3. *Coarse approximations* of nonlinear activation functions.

To the best of the author's knowledge, up until now, the effect of approximate computing on LSTMs have not been investigated. In the rest of this chapter, a section is dedicated to each of these three approaches. The rationale behind each one is explained, experiments are conducted and results are presented.

5.3 Word length and fixed-precision arithmetic

5.3.1 Numerical Types

In order to implement any algorithm, all input data, parameters, coefficients and intermediate variables and states must be represented with a specific numerical type [33]. These representations of numbers play a major role on the behaviour of the

algorithm, and thorough understanding of their advantages and limitations is crucial for correct system design. For each numerical representation, there are two important properties, *dynamic range* and *precision*.

Precision is the smallest increment that can be represented in a numerical type and is defined as:

$$d = 2^{-n} \tag{5.1}$$

where:

n = the number of fractional bits used in the representation.

Dynamic Range is defined as:

$$D = \frac{V_{max} - V_{min}}{d} \tag{5.2}$$

where:

V_{max} = the largest value that can be represented
V_{min} = the smallest value that can be represented
d = the smallest increment of V, i.e. the precision.

Different programming languages and platforms make different assumptions of the representation of data. In this work, the implementation and training of the LSTM/FC architecture in the Torch framework assumed the double-precision floating-point format as specified by the IEEE 754 standard since Torch is based on a Lua/C++/CUDA implementation. This standard specifies 1 bit for as the sign bit, 11 bits for the exponent and 53 bits (52 explicitly stored) for the significand (or mantissa), therefore occupying a total of 64 bits.

The real value assumed by a given datum in this format can be expressed as

$$\tilde{V} = Q(V) = (-1)^{sign}\left(1 + \sum_{i=1}^{52} b_{52-i}2^{-i}\right) \times 2^{e-1023} \tag{5.3}$$

where:

$sign$ = the sign bit
b_i = binary digits $b_i = 1, 0$
e = the exponent.

The large number of bits and the mantissa-exponent representation guarantee both very wide dynamic range and an adequate precision for most applications, including training and evaluating the LSTM/FC architecture of Chapter 4. However, its performance and bandwidth cost remain significant, therefore motivating an investigation towards using different, simpler and less costly numerical representations, which will still guarantee the precision and range necessary for the LSTM/FC forward pass while significantly reducing computational, bandwidth and energy costs.

When implementing in hardware, it is important to remember that shorter word lengths always result in less power consumption [33], smaller chip area and faster computation. Also, switching from floating-point arithmetic to fixed-point results in many hardware simplifications that can also lead to a smaller energy footprint.

When representing a real value in a fixed-point format, the values is represented by a weighted sum of bits.

Using a slope and bias encoding scheme, a fixed-point signed number representing the real value can be expressed as

$$\tilde{V} = Q(V) = S \cdot \left[-b_{WL-1}2^{WL-1} + \sum_{i=0}^{WL-1} b_i 2^i \right] + B \tag{5.4}$$

where:

b_i = binary digits $b_i = 1, 0$ for $i = 0, 1, \ldots, wl - 1$
WL = the word length
S = the scaling factor
B = the bias

while the range that the real value can span is restricted in

$$\left[S \cdot \left(-2^{WL-1} \right) + B, S \cdot \left(2^{WL-1} - 1 \right) + B \right] \tag{5.5}$$

As it can be seen, the range difference for a fixed-point and a floating-point number or the same word length vary by many orders of magnitude.

5.3.2 Considerations

Overflow

When dealing with double precision numbers in real-world applications, exceeding the dynamic range is rarely an occasion; overflows or underflows are much more common in fixed-point arithmetic, where the usable dynamic range is much narrower. It should be noted that even if all input and output data of an algorithm are guaranteed to lie in the range of fixed-point type, there might be intermediate results (such as counters, partial products, temporary variables, etc.) that might exceed the expected range, causing an overflow or underflow and leading to aberrant behaviour.

Rounding Errors

Arithmetic calculations with fixed-point types usually demand a different word length to store their result. For example, the addition of two N bit numbers requires N+1 bits to safely store the result while the multiplication of an M bit and an N bit number requires up to M+N bits. If these calculations are to be applied iteratively, rounding or truncations should be applied in order for the word length to not increase infinitely. This raises concerns about the rounding methods applied and the quantization error that they will introduce.

5.3.3 Experiments

In this work, we focus on the implementation of the LSTM/FC architecture on a tailored, embedded hardware architecture with minimal energy footprint. Only the inference stage of the design is dealt with, as the training, testing and fine-tuning of the architecture's parameters are completed beforehand using double precision in an environment where time and energy consumption are not considered constraints.

The inference stage (i.e. the forward pass) is reimplemented in MATLAB and quantization to different word lengths is applied to the network's parameters (weights and biases), input data (MFCC sequences) and internal variables and states.

Quantizing a set of data to a fixed point format is not a trivial task; Allowing too many bits for the integer part of a word will result in loss of precision, while allowing too many bits for the fractional part significantly limits the representable dynamic range. Depending on the *statistics* of the data to be quantized, there is an *optimal solution* that adequately represents the minimum and the maximum of the data and while retaining as much precision as the chosen word length allows.

Since the parameters of the LSTM/FC network (all weights and biases $\mathbf{W}, \mathbf{R}, \mathbf{b}$ for all three blocks) are known beforehand, they are considered as one distribution and they are quantized separately from the MFCC input data. The MFCC data are quantized to the same word length, but are considered as a different distribution and therefore quantized with different scaling than the parameters.

To quantize parameters and input data to a word length WL, the following procedure is followed:

1. Find the next power of 2 that specifies a range that fits the maximum absolute value of the data;
2. Scale this range in the $(-1, 1)$ interval;
3. Multiply the scaled data with 2^{WL};
4. Round to the nearest integer;
5. Divide with 2^{WL} ;
6. Scale back to the range specified in 1.

Two different cases are also examined: One where only the LSTM parameters are quantized and one were both the LSTM and the fully connected layer parameters are quantized.

For each case, the LSTM/FC is evaluated over the validation set of TIMIT. The above procedure is implemented in pseudocode and can be seen in Algorithm 5 and the full implementation can be found in appendix A.2.

The results of these experiments are reported as the relative loss of accuracy:

$$\text{Relative Accuracy} = \frac{\text{Accuracy achieved in this experiment}}{\text{Baseline accuracy}} \tag{5.6}$$

Algorithm 5 Fixed Precision Emulation

1: **for all** parameters $\mathbf{W}, \mathbf{R}, \mathbf{b}$ **do**
2: Quantize $w$to the specified word length WL
3: **end for**
4: **for** every input vector $\mathbf{x_t}$ **do**
5: Quantize $\mathbf{x_t}$ to the word length WL
6: Run the LSTM/FC forward pass
7: Output a prediction vector $\mathbf{y_t}$
8: **end for**
9: Evaluate performance

Figure 5.1 shows the relative deterioration of accuracy as the number of bits used for the representation of the LSTM/FC parameters is reduced. It is apparent that migrating from double-precision floating point using 64 bits to single-precision floating point using 32 bits no deterioration is observed. This leap in the number of bits significantly reduces the computational requirements for the algorithm, requiring less hardware. Another advantage is that it halves the memory bandwidth needed for the fetch of the architecture's parameters from the main memory.

Switching from 32-bit floating point to 16-bit fixed-point and quantizing the parameters, the inputs and all intermediate variables to that format still shows no penalty in predictive performance. This step allows for fixed-point hardware.

It appears that reducing the word length as far as 8 bits has no effect on the network's performance, while 7 bits drive relative accuracy to 96%. The combination of this behaviour with the fact that LSTM networks have a *small number of parameters* relative to other, similarly-performing architectures that don't incorporate temporal context natively (see 2.4.1) proves that LSTM based architectures on approximate computing platforms are *prime candidates* for embedded and mobile Automatic Speech Recognition systems and other sequential machine learning tasks.

5.3.4 Conclusions

In this section, the advantages and considerations of using numerical representations with a small amount of bits were discussed. When migrating to shorter word lengths one must proceed with care, since overflows, limited precision and other factors may lead to aberrant behaviour of the system under test. The experiments that were conducted show no loss in classification accuracy even when using a 8-bit fixed point representation while 7 bits drive relative accuracy to 96%.

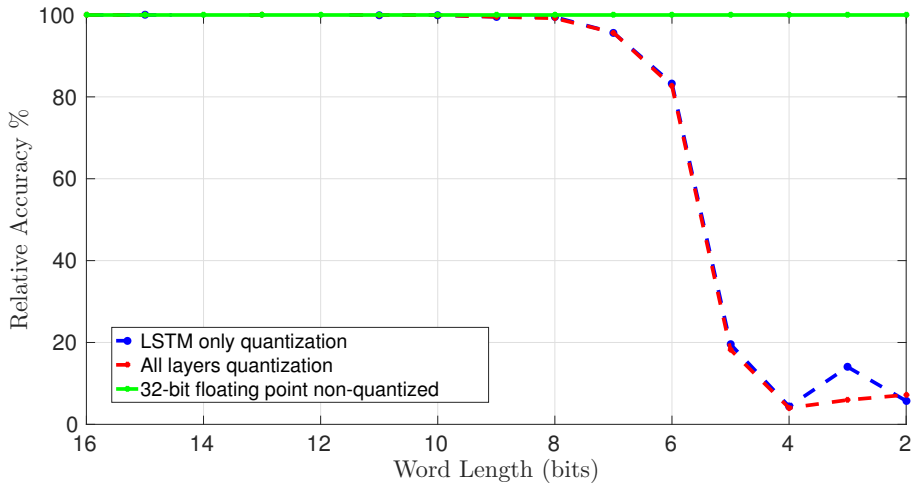


Figure 5.1: The effect of quantization on accuracy. Quantizing the previously 64-bit floating-point numbers to 32 bits shows no loss in relative accuracy. Quantizing to even shorter word lengths shows that there is no loss with representations using up to 8 bits, while 96% relative accuracy can be retained even with 7 bits. Quantizing even more results to drastic reduction of the LSTM/FC accuracy.

5.4 Connection pruning and sparse data

5.4.1 Motivation

In 4.6.3 the statistical properties of the learned parameters were documented. Intuitively, one can understand that the larger the absolute value of a weight, the stronger the connection between two neurons is, and accordingly, the more "important" the connection for the process of inference. From a statistical point of view, this points to the fact that neural network performance and behaviour is governed by the outliers in the network parameter distribution.

Based on that observation, it is worth investigating the pruning of all connections under a specific "importance" threshold, since they can be regarded as redundant. From a data point of view, this equals the zeroing of all weight values under a numerical threshold. This operation is equivalent with artificially increasing the sparsity of the matrices, and conceptually corresponds to decreasing the coupling among the system's basic functional units (neurons).

A sparse matrix is a matrix where a big amount of its elements are zero:

$$\begin{bmatrix} 0 & 0 & 3 & \dots & 0 \\ 1 & 0 & 0 & \dots & 0 \\ 0 & 4 & 0 & \dots & 2 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 8 & 0 & 11 & \dots & 0 \end{bmatrix}$$

and the sparsity of a matrix is defined as

$$\text{Sparsity} = \frac{\text{Number of zero-elements}}{\text{Total number of elements}} \tag{5.7}$$

5.4.2 Experiments

In the experiments of this section, connection pruning was simulated by increasing the sparsity of architecture's parameter matrices. Since sparsity is desired as the independent variable and the networks parameters don't follow a uniform distribution, an adaptive thresholding method was implemented so as to select and zero the exact amount of elements needed to meet a specific sparsity. In this way, better control on the trials was obtained. The pseudocode in Algorithm 6 describes how the connection pruning is performed, while the full implementation can be found in appendix A.2.

Algorithm 6 Sparsity Emulation

1: **for all** parameter matrices $\mathbf{W} \in \{\mathbf{W}, \mathbf{R}, \mathbf{b}\}$ **do**
2: Take $|\mathbf{W}|$
3: Find the desired sparsity percentile P ▷ The desired percentage of zero elements
4: Set all elements of $\mathbf{W}$ below P to 0 ▷ This effectively prunes weak connections
5: **end for**
6: **for** every input vector $\mathbf{x_t}$ **do**
7: Run the LSTM/FC forward pass
8: Output a prediction vector $\mathbf{l_t}$
9: **end for**
10: Evaluate performance

Three experiments are performed; In the first, only the connections in the two LSTMs are pruned. In the second, only the fully connected layer's parameters are pruned and in the third, connections both from the LSTM blocks and the fully connected layer are pruned. It is interesting to observe that pruning only the fully connected layer has the same effect on relative accuracy as pruning the LSTM blocks. However, the fully connected layer parameters amount to only 8% of the total number of parameters, as can be seen in Table 4.1. Therefore, there is incentive *not to prune* the fully connected layer.

Since the LSTM is on its own a wired collection of small fully connected layers and the architecture has two LSTM blocks and a fully connected layer, it is apparent that the search space for an optimal sparsity solution is quite large. More specifically, the number of layers to "sparsify" is quite large, therefore the dimensionality of this optimization problem is high. In the future, this sparsity analysis will be performed more thoroughly along with the necessary statistical analysis of variance to quantify the importance of each layer to the overall design and to discover any underlying correlations.

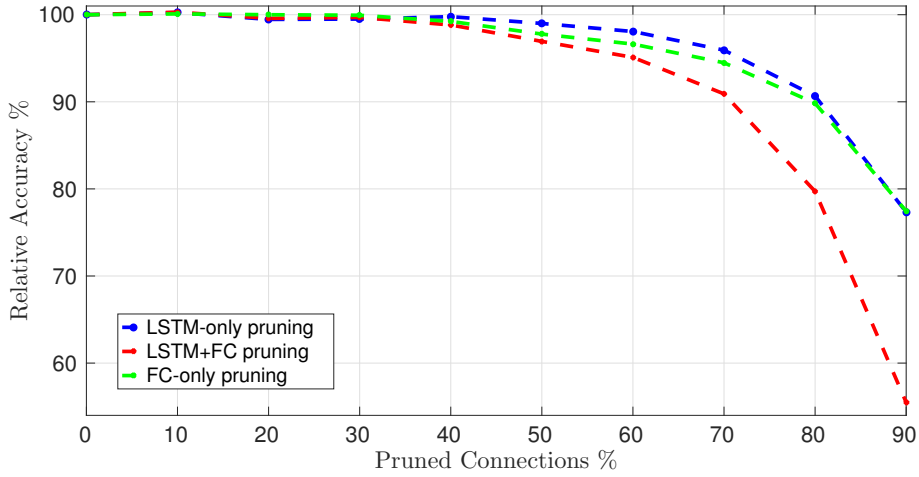


Figure 5.2: The effect of connection pruning on accuracy. By zeroing elements of the weight matrices with absolute values under a threshold, we can implement connection pruning. Pruning up to 40% of the LSTM connections brings no penalty in relative accuracy, while pruning 70% of the LSTM connections results in 96.1% relative accuracy. Pruning only the fully connected layer shows similar results, while pruning both shows larger performance degradation.

Figure 5.2 shows how connection pruning of layers influences overall network performance. Pruning only the LSTM layers, 40% of pruned parameters leads to no accuracy loss, while even with 70% of the parameters set to zero, relative accuracy remains at 96.1%. Pruning only the fully connected layer leads to similar results, but pruning both leads to significant deterioration of accuracy.

From a hardware point of view, exploiting this behaviour correctly can significantly reduce memory bandwidth usage by avoiding costly DRAM fetches of all-zero words. Relying on a lossless compression scheme and a processor-side decoding procedure, the encoded words can be efficiently fetched and decoded in the processor. After the fetch, energy consumption can be reduced even more through avoidance of zero product calculations, with the checks implemented either in software or hardware.

5.4.3 Lossless compression

The analysis on the effect of word length on network performance described in 5.3 shows that reducing the number of bits required to store the network's parameters is, up to some extent, a very effective way of lowering the architecture's memory bandwidth requirements and therefore its energy consumption. This quantization to narrower word lengths can be though of as a form of lossy compression where network performance degrades after a specific threshold.

Increasing the sparsity of the network's parameters (zeroing out near-zero values) has the effect of lowering their total *Shannon entropy*, allowing for the efficient application lossless encoding schemes that can reduce the total amount of memory needed

Word Length	Pruning	Huffman Encoding	Size (B)	Compression
16	0	no	375,418	
8	0	no	187,709	2.0×
8	0	yes	106,421	3.5×
8	40%	yes	86,678	**4.3**×
8	70%	yes	55,158	**6.8**×
7	0	no	164,245	2.3×
7	0	yes	83,329	4.5×
7	40%	yes	74,740	**5.0**×
7	70%	yes	48,540	**7.7**×
4	0	no	93,854	4.0×
4	0	yes	35,639	10.5×
4	40%	yes	31,495	11.9×
4	70%	yes	27,955	13.4×

Table 5.2: Parameter encoding and compression. After quantization and pruning is enforced on the data, there is opportunity for lossless encoding that can drive the size of data that need to be transferred from the memory to the processor even more down. For a 16-bit baseline, quantizing to 8 bits leads to 2× lossy compression, which can be augmented losslessly to 4.3× with 40% pruning with no loss in predictive accuracy. Allowing for 96.1% relative accuracy gives the opportunity of compressing up to 6.8×.

to store the data. This, along with an efficient software/hardware implementation of a decoder, can further reduce the necessary memory bandwidth needs of the machine learning architecture.

Due to the nature of the calculations in neural networks and especially Fully Connected NNs, it can be shown that there is small if any data reuse of the network parameters for each individual input. This implies that the data that are fetched from the DRAM to the cache memory should be available for use in calculations and then discarded, freeing space for the next DRAM fetch. Under this consideration, schemes like Run-length encoding that operate on runs of data instead of individual data elements should be used with care, since they would require extra control structures to deal with data that were either fetched when it was unnecessary or vice-versa.

In this work, Huffman coding is applied to the network's parameters after they have been quantized to a specific word length and after undergoing sparsification. The results on total size can be shown in the Table 5.2.

We can see that for a 70% sparse set of network parameters, we can achieve up to 3.3× lossless compression over the 7-bit quantized parameters.. Including the lossy compression induced by the quantization, this can amount to up to 7.7× compression. Since DRAM load/store operations are one of the major energy consumption bottlenecks, this approach can significantly reduce the energy footprint of a machine-learning-optimized hardware architecture. It should be noted, however,

that increased arithmetic and control operations would be needed in the processor, in order decode the incoming compressed stream of parameters.

5.4.4 Conclusions

In this section, based on observations from 4.6.3 about the statistical properties of the LSTM parameters, the effect of connection pruning on the LSTM parameters was investigated. Experiments that prune unimportant connections from both the LSTM and the FC layers were conducted, and it is shown that the system's accuracy suffers no penalty for even up to 40% pruning, while allowing for 96% of relative accuracy allows for up to 70% connection pruning. This fact can be used to design architectures that save energy a) by lossless compression of the parameters (and therefore fewer memory accesses) and b) by skipping unnecessary computations with zero-valued parameters.

5.5 Coarse approximations of the activation functions

5.5.1 Motivation

During the architecture's forward pass the activation function $\sigma(x)$ and the hyperbolic tangent $\tanh(x)$ (shown in Figure 5.3) are applied to many points of the design. The sigmoid function is defined as:

$$\sigma(x) = \frac{1}{1 + e^{-x}} \tag{5.8}$$

and its first order derivative is defined as:

$$\frac{d}{dx}\sigma(x) = \sigma(x)(1 - \sigma(x)). \tag{5.9}$$

The sigmoid is a generalization of the logistic function

$$f(x, L, \alpha, x_0) = \frac{L}{1 + e^{-\alpha(x - x_0)}} \tag{5.10}$$

where:

e = the base of the natural logarithm
L = the curve's maximum value
α = the steepness of the curve
x_0 = the x-value of the curve's midpoint.

The hyperbolic tangent function is defined as:

$$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} \tag{5.11}$$

with its first order derivative defined as:

$$\frac{d}{dx}\tanh(x) = 1 - \tanh^2(x) \tag{5.12}$$

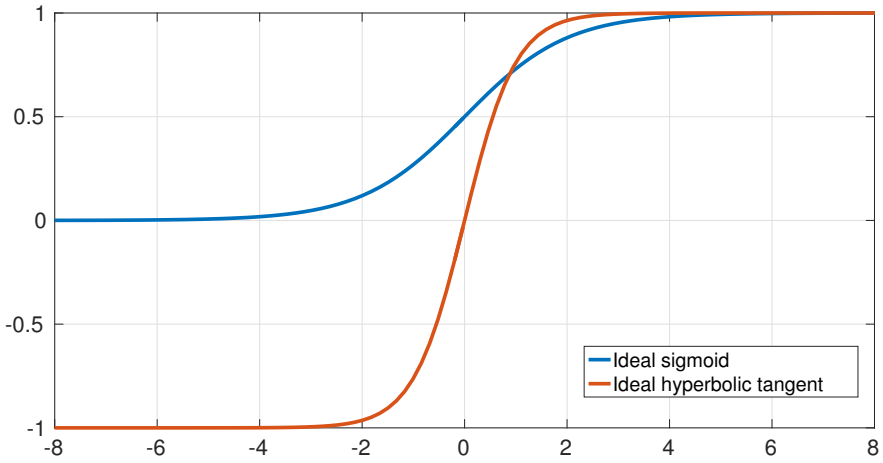


Figure 5.3: The activation functions. The ideal sigmoid and hyperbolic tangent will be approximated with lookup table implementations because these allow for faster, cheaper implementations.

The $\tanh(x)$ is related to $\sigma(x)$ with the following equation:

$$\tanh(x) = 2\sigma(2x) - 1 \tag{5.13}$$

These two function are largely equivalent, since the linear transform of equation 5.13 implies that any function computed by a neural network with the sigmoid as an activation function can be computed with another network with the tanh as an activation function[11].

The reason that LSTM networks use both has to do with their output range; The sigmoid output range is $(0, 1)$ while the tanh range output range is $(-1, 1)$. The calculation of the candidate state $\tilde{\mathbf{C}}_\mathbf{t}$, for example, uses the tanh activation function because the network needs to be able to *remove* (i.e. subtract) information from the previous cell state $\mathbf{C_{t-1}}$. An excellent study that assesses the performance of LSTM networks when different modules are added, removed or swapped with others can be found in [15].

In the LSTM/FC architecture, these activation functions are applied at the four gates $\mathbf{f_t}, \mathbf{i_t}, \mathbf{o_t}, \tilde{\mathbf{C}}_\mathbf{t}$ and also at the computation of the output $\mathbf{h_t}$. All the applications of these nonlinear functions are applied element-wise to vectors of size h. Therefore, in the current architecture which has $h = 128$ and 2 LSTM blocks, for each input vector $2 * (5 * h) = 1280$ nonlinear activation function evaluations must take place.

It is therefore clear that the hardware-specific implementation of these functions can influence the overall performance of the system. Although there are many methods of both software and hardware implementations of function approximations (CORDIC, power-series and polynomial approximations), table-lookup algorithms (LUT) have proven to be superior in speed and accuracy, while allowing for controllable precision, tight error analyses and predictable error bounds. Also, implementing

with table-lookup is effectively *function independent*; Whether the function to be evaluated is the tanh or the sigmoid makes no difference in the implementation. Finally, table-lookup algorithms are easily and cheaply implemented in systems that feature hardware multipliers.

5.5.2 Computational complexity of LUT approximations

Requirements

Approximating a smooth function $f(x)$ over a domain of interest I with a LUT-interpolation approach requires, in general:

1. a vector X containing N discrete and preselected values in I, called breakpoints
2. a vector Y containing N corresponding values in the image of f, specifically $f(x)|x \in X$
3. an interpolation mechanism that can map a desired value $x_d \in I$ to a value $\tilde{f}(x_d)$.

Linear Interpolation

A linear interpolation mechanism that maps a desired value x_d to a value $\tilde{f}(x_d)$ is defined as follows:

$$\tilde{f}(x_d) = \frac{f(x_i) - f(x_{i-1})}{x_i - x_{i-1}}(x_d - x_{i-1}) \tag{5.14}$$

where

x_i, x_{i-1} = breakpoints, $\in X$
$f(x_i), f(x_{i-1})$ = precomputed values of f, $\in Y$

From a computational point of view and depending on the spacing between breakpoints, the implementation of this mechanism can vary in complexity.

Breakpoints and their spacing

Increasing the number of breakpoints N, i.e. the points where the ideal function is precomputed and stored, can have the advantage of resulting to finer approximations, while increasing the necessary memory space for the storage of the precomputed values. However, the spacing between breakpoints plays a more significant role in better approximating the desired function, while it introduces more concerns about its hardware performance.

There exist, in general, three ways of spacing the N breakpoints, each with its advantages and disadvantages.

1. **Even spacing**. Placing the N breakpoints evenly in the domain of interest I allows for the omission of an explicit declaration like $X = [x_1, x_2, x_3, \ldots, x_n]$ and only requires the declaration of the breakpoints' evaluated values $Y =$

$[f(x_1), f(x_2), f(x_3), \ldots, f(x_n)]$. This halves the amount of memory space needed and also simplifies how the lookup table will determine where among the breakpoints the desired input x_d lies.

2. **Power-of-2**. Fixing the N breakpoints on 2^n values also allows for the omission of X, while also simplifying the necessary calculations during the linear interpolation. However, this method will introduce higher error for a specific N, since approximating functions with non-uniform curvature like in this case, $\sigma(x)$ and $\tanh(x)$, requires more breakpoints to achieve the same amount of accuracy. Therefore, a larger N is expected to be needed.

3. **Uneven spacing**. Placing the N in an uneven fashion in the domain of interest I necessitates the explicit declaration of both the X and the Y vector in memory storage. This method has the advantage that it requires the fewer points to achieve minimum error, since breakpoints can be placed "strategically", based on the curvature of $f(x)$. However, this method requires a binary search algorithm (assuming the vector X is sorted) than will execute at maximum $\log_2(N)$ times to determine where the input x_d lies amongst the breakpoints. Moreover, this method will require the processor to execute all the necessary multiplications and divisions needed to linearly interpolate $\tilde{f}(x_d)$ from equation 5.14.

Maximum absolute error

When approximating an ideal function with a lookup table function, the error at any point x_d in I is defined as:

$$\epsilon(x_d) = |f(x_d) - \tilde{f}(x_d)| \tag{5.15}$$

and the maximum absolute error is defined as:

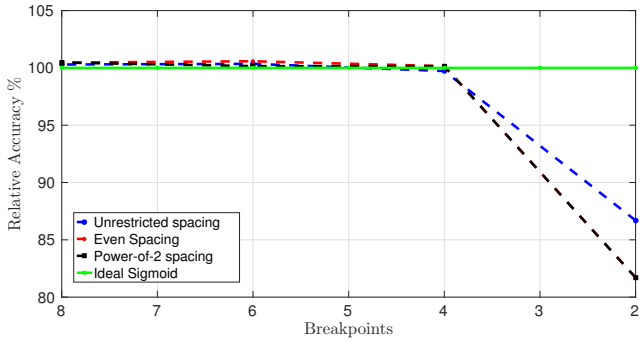
$$\epsilon_{max} = \max(\epsilon(x_d)) \quad x_d \in I \tag{5.16}$$

The placing of the breakpoints greatly influences the error, and therefore there is a trade-off between simplicity in the hardware/software implementation (discussed above) and absolute error.

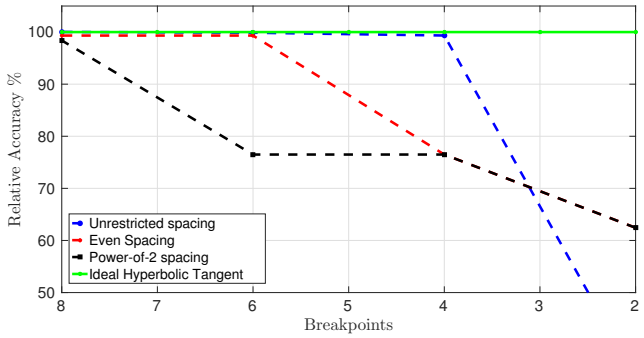
5.5.3 Experiments

To evaluate the feasibility of a table-lookup approximation for the nonlinearities present in neural networks, a MATLAB emulation took place, where the ideal sigmoid and hyperbolic tangent functions described in 5.5 are replaced by table-lookup versions, where their respective domain of interest I is split in $N + 1$ parts, and the ideal functions are evaluated beforehand only at N breakpoints.

Since the nature of both $\sigma(x)$ and $\tanh(x)$ is limiting (at their extrema, they infinitely compress their inputs to 0 or 1 and to -1 or 1), the behaviour of the emulated



(a) Sigmoid LUT approximation



(b) Hyperbolic Tangent LUT approximation

Figure 5.4: The effect of coarse activation functions on accuracy. The three cases evaluated are *unrestricted breakpoint spacing* (blue), *even spacing* (red), *power-of-2 spacing* (black). In (a) it is shown that replacing the sigmoid function even with the cheapest, easiest to implement *power-of-2* approximation with 4 breakpoints, no loss in accuracy is observed. In (b), approximating $\tanh(x)$ with the same power-of-2 strategy requires 8 breakpoints to achieve perfect relative accuracy.

functions extrapolates values x outside the domain of interest I to $\lim_{x\to\pm\infty} f(x) = c$ while the absolute error approaches zero:

$$\lim_{x\to\pm\infty} \epsilon(x) = 0$$

The approximate functions are used in the machine learning architecture and the effect on relative accuracy is evaluated. With this method, the robustness of the LSTM/FC design is effectively evaluated against the error that can occur when coarsely approximating the ideal functions with their LUT-interpolated versions. The results are shown in Figure 5.4.

5.5.4 Conclusions

In this section, different ways of approximating the non-linear activation functions used in the LSTM architecture are examined in terms of approximation error and in

terms of computational complexity. It is shown that breakpoint placing critically influences both these factors. Furthermore, approximate versions of both the sigmoid and the hyperbolic tangent functions are used in the machine learning architecture, and it can be seen that even with the cheapest and simplest approximations, almost no accuracy loss is observed. This analysis validates the previous assumptions on neural network error resilience and allows for an informed, LUT-based implementation of the activation functions that is computationally simple without sacrificing performance.

5.6 Conclusion

In this chapter, an analysis on the error resilience of LSTM networks was performed. Similar analyses have been documented for fully connected neural networks and other architectures like convolutional neural networks, but an analysis of this kind for LSTM networks had not yet been performed.

The investigation followed three directions:

1. The effect of quantization of LSTM parameters and data was addressed, and it proved that LSTMs suffer no accuracy loss even when using 8 bits for their parameters and calculations.

2. The effect of connection pruning was addressed, and it proved that no accuracy loss occurs even with 40% of their connections pruned, while allowing for a 4% relative accuracy reduction allows for 70% pruning.

3. The non-linear computationally expensive activation functions were replaced with look-up table approximations, proving that no accuracy loss is observed even when the lookup tables contain as few as 8 precomputed values. Different interpolation mechanisms were also assessed, to ensure that the interpolation computations wouldn't nullify the effects of the original decision to use LUT-based approaches. It is shown that the easiest to implement interpolation mechanism that uses power-of-2 spaced breakpoints achieves perfect relative accuracy with 4 breakpoints for $\sigma(x)$ and 8 points for $\tanh(x)$.

The above results motivate the proposition of a hardware architecture that can exploit them in order to achieve energy savings. A modified ASIP architecture, computational kernels and an energy model will be presented in the following chapter.

Chapter 6

A Hardware Implementation

6.1 Introduction

In modern SoCs, designers integrate general-purpose processors with many hardware accelerators for different system functions, each specifically designed to be tailored to the needs of the applications that requires it. Modems, GPUs, Video/Audio encoding/decoding, all are designed to achieve the best combination of performance, energy consumption, cost and design time. This traditional ASIC model, however, is currently undergoing a complexity crisis. This has started a shift in design methodology, where designers seek programmability and the ability to reuse the same platform with a minimum amount of changes, minimizing time-to-market and other non-recurring engineering costs such as ASIC design starts and functional verification costs.

Field-Programmable Gate Arrays (FPGA) have been praised for their ease of programmability and configurability, but due to their power inefficiency they usually just serve as rapid-prototyping platforms rather than complete, market-ready solutions. Hence, there appears to be a spectrum gap between high flexibility provided by general-purpose processors and high performance provided by fixed-functionality accelerating hardware.

Application Specific Instruction Set Processors (ASIP) fill the aforementioned gap by allowing the implementer to intervene and design in an abstraction level that allows both for high efficiency (through low energy consumption and high throughput) and high flexibility through programmability.

6.2 Application Specific Instruction Set Processors

In ASIP design, the designer can focus on modifying or expanding the instruction set of a microprocessor in order to optimize performance for a given application. Also, the designer can customize the datapath to exploit parallelism inherent to the application. Combining these two points i.e. creating instructions that utilize a custom, efficient datapath tailored to the application results in software programmable processors that natively exhibit aptitude or "talent" to a specific task. In that way, high-level

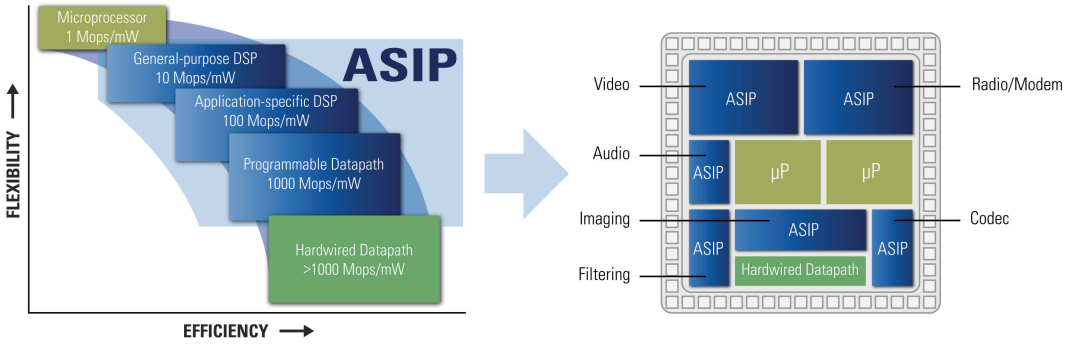


Figure 6.1: Application Specific Instruction set Processors (ASIP). ASIPs are more flexible than a hardwired datapath present in an ASIC. This allows for fast customization and programmability and shorter time-to-market. However, it also allows for efficient processing and low energy consumption.

programming (e.g. C/C++) can be used to rapidly develop an implementation that will exploit the architectural details of the ASIP to achieve the highest possible performance for a given energy budget or vice-versa.

The first step in ASIP design is architecture exploration. *Amdahl's law* states that the application's bottlenecks are the parts that need to be optimized in order to attain improvements. With that in mind, specific cycle-accurate profiling tools must be used to identify which parts of the application execute more often, what types of instructions are most frequently used, if sets of instructions are always appearing as sets (perhaps signifying an optimization opportunity) or if specific functions or procedures monopolize execution time.

Next, the instruction set architecture (ISA) should be defined, and the micro-architecture that implements the defined ISA should be designed. In this part of the design flow is where the optimization opportunities and their exploitation arise.

Architectural implementation is the next step in ASIP design, and it involves deriving an HDL model from the architecture description. After a synthesizable HDL model (such as VHDL or Verilog) has been derived, the standard synthesis flow can be followed. There, simulation in the RTL level can be used to examine resource usage and energy consumption, while manufacturing technology constraints can also be enforced.

The design of applications for the ASIP is also part of the ASIP design methodology, and so is system integration and verification. In this final step, the newly designed ASIP is co-simulated along with other System-on-Chip components to ensure correct and efficient behaviour.

6.2.1 Observations

This section describes two major observations that differentiate ASIP design from other types of hardware design.

Tool-chain in the loop

In ASIPs, the ISA and the related microarchitecture can be hand-designed to meet specific performance criteria. However, in order to maintain High Level Language compatibility, each designed ASIP should come with the necessary tools for the development of applications. The absolute basic Software Development Kit for any microprocessor should consist of an optimizing compiler, a linker, an assembler, a simulator and a profiler. However, manually redesigning these tools is error-prone and time-consuming and this process would probably outweigh the advantages of ASIP design altogether.

For this reason, retargetable toolchains should be implemented, that *mutate* along with the ASIP under design, exploiting its low-level optimizations to create more efficient application code. The automatic generation of an SDK is a very difficult task on its own, and that is why a necessary level of formalism is required when designing a custom ISA and its microarchitecture.

Architecture description formalism

To facilitate efficient ASIP design *architecture description languages* (ADLs) have emerged. ADLs describe the processor on a higher abstraction level, and bridge the gap between High Level Languages (HLLs) and Hardware Description Languages (HDLs). The formalism introduced by ADLs has two main contributions; First, it allows for the automatic generation of the toolchain needed by the application developers and second, it allows for the generation of a synthesizable HDL model that can be further simulated and implemented.

6.3 The LSTM/FC architecture on an ASIP

In section 4.1 we reasoned that from a hardware point of view, the main advantage of utilizing neural network architectures for machine learning is their regular and homogeneous computations. The LSTM, being a type of neural network, also exhibits very predictable behaviour computationally. In Figure 6.2 the LSTM forward pass is presented as a diagram of operations. It is easy to see that minimal control is needed for the execution of the forward pass; There are no conditional jumps or breaks, and each input $\mathbf{x_t}$ produces exactly one output $\mathbf{h_t}$, while altering one internal state variable, $\mathbf{C_t}$.

Therefore, an ASIP that would efficiently perform the LSTM forward pass should have the following features:

- Perform Matrix-Vector multiplications in the least amount of cycles possible;
- Perform Vector-Vector additions and multiplications in the least amount of cycles possible;
- Evaluate nonlinear functions in the least amount of cycles possible;
- Have a word length that is just enough for no loss in LSTM accuracy;

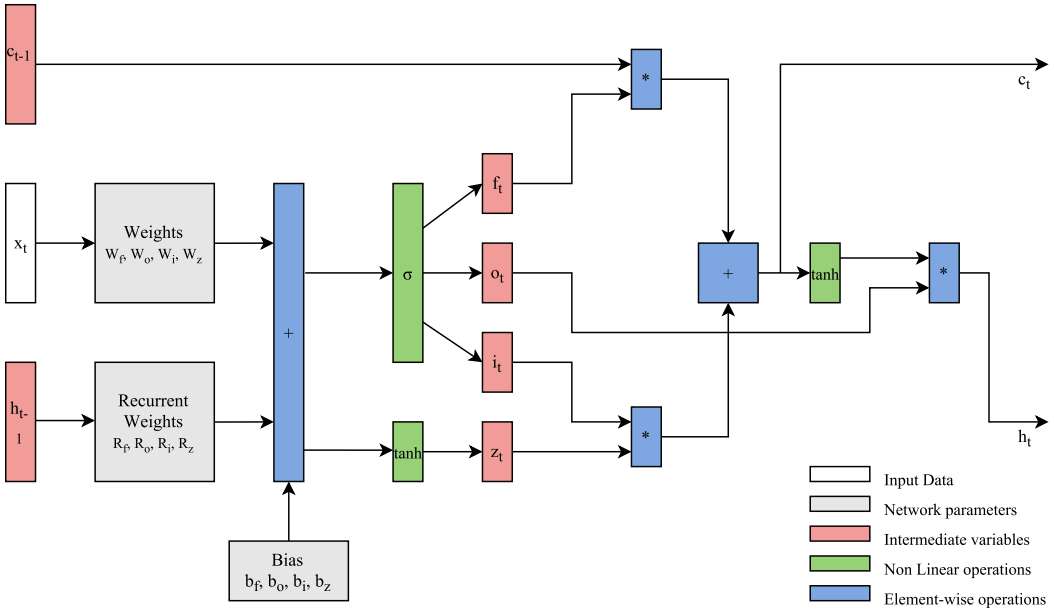


Figure 6.2: A computational approach to the LSTM. Seen from a computational/algorithmic perspective, the LSTM is a combination of matrix-vector multiplications (gray), element-wise vector operations such as additions and multiplications (blue) and element-wise applications of nonlinear activation functions on vectors (green). For each incoming MFCC frame $\mathbf{x_t}$ (in white), the previous cell state $\mathbf{C_{t-1}}$ and output $\mathbf{h_{t-1}}$ (in red) take part in the computations to calculate a new output $\mathbf{h_t}$ and cell state $\mathbf{C_t}$.

- Skip unnecessary computations.

In the following section, each of these points are addressed in various levels of abstraction, since this is a core advantage of ASIP design.

6.3.1 Matrix-Vector multiplications

In order to achieve minimal cycle count for the computation of all matrix-vector products of the general form

$$\mathbf{y} = \mathbf{A}\mathbf{x} \quad x \in \mathbb{R}^d, y \in \mathbb{R}^h \tag{6.1}$$

$$y_i = \sum_j A_{ij} x_j \tag{6.2}$$

two approaches are used.

SIMD

Firstly, a Single Instruction Multiple Data (SIMD) vector processing unit of parameterizable size VSIZE is utilized to

1. Read VSIZE elements of **A** in one cycle into the vector register file
2. Read VSIZE elements of **x** in one cycle into the vector register file
3. Evaluate VSIZE partial products $A_{ij}x_j$ in one cycle
4. Evaluate the sum of VSIZE partial products in one cycle
5. Accumulate the result in a register.

With corresponding SIMD instructions that utilize the vector processing unit, the above operations can be performed in parallel on data that are contiguous in memory and complete in one clock cycle. Data locality and layout are critical points and were considered during the implementation. All data were aligned to a multiple of $VSIZE * wordlength$ in memory by enforcing constraints on their allocation.

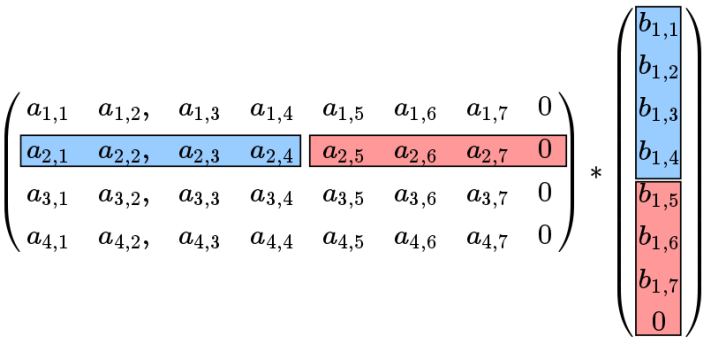


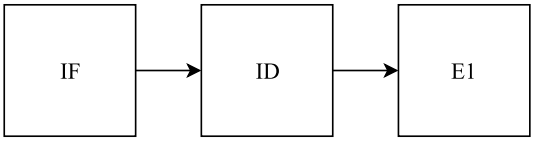
Figure 6.3: SIMD and Memory Accessing. In order to achieve best throughput the data in the memory should be padded to an integer multiple of the SIMD size. In this way, no control needs to be implemented.

One other difficulty that emerges in data layout is the phenomenon of edge effects; Code that can operate on data vectors with sizes not integer multiples of the SIMD size VSIZE will usually be very control heavy and will possibly nullify the benefits of SIMD processing altogether. The solution that is enforced for this challenge is *zero-padding*, i.e. padding a vector of size N with M zeroes so that $(N + M)$ mod VSIZE $= 0$. By adhering to this principle, all code can be check-free with respect to edge effects and therefore greatly simplified in terms of control flow. The overall result is not altered by this padding, since zero has both the multiplication and the addition property.

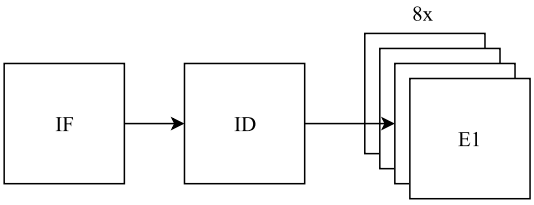
The computational kernels for matrix-vector multiplication that exploit this hardware structure are shown in Appendix A.3.3.

Pipelining

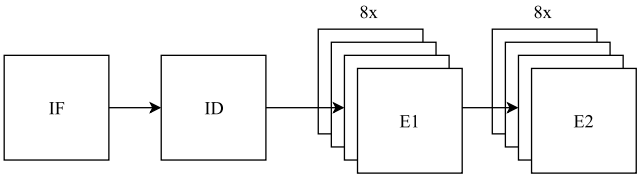
The machine code generated for the parallel loop will be looped $\frac{h*d}{\text{VSIZE}}$ times to complete the overall matrix multiplication. Since there are no conditions that can



(a) SISD processor with three pipeline stages



(b) SIMD processor with three pipeline stages



(c) SIMD processor with four pipeline stages

Figure 6.4: Pipeline Stages. In (a), a simple diagram of a SISD pipeline stage is shown. In (b) the E1 stage of the pipeline is vectorized, allowing the processor to process 8 data elements under one instruction. In (c) a new stage E2 is added to the processor, allowing for further parallel processing. The deepening of the pipeline adds latency to the processor, but allows more throughput, since now stage E1 can process a next set of data elements while E2 is processing the previous.

alter the flow of execution, it is possible to define extra pipeline stages into the datapath of the architecture to enable a multi-stage instruction than has higher latency but allows for more throughput, thus reducing the amount of clock cycles needed for the completion of a matrix-vector multiplication.

Figure 6.5 shows the pipelined architecture, while in Appendix A.3.1 the nML code that implements the pipelined instructions is shown.

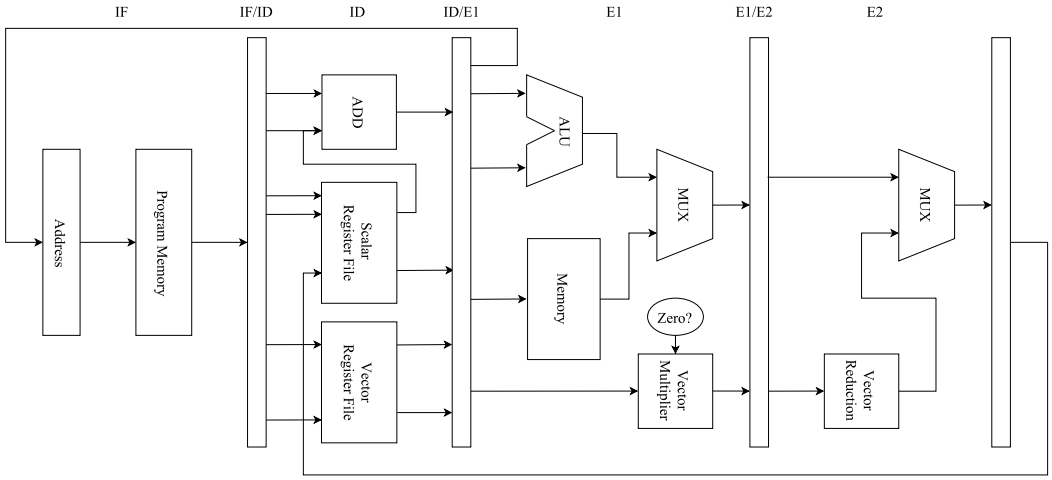


Figure 6.5: The modified ASIP architecture. This architecture has the following modifications: a) It incorporates a vector register file and a vector multiplier that has hardware checking to avoid switching if any operand is zero and b) It has an extra pipeline stage E2 where the vector reduction is executed before it is written back in the register file. With this structure, a vector-vector inner product can be achieved in exactly one clock cycle, allowing for acceleration of the matrix-vector product process.

Figure 6.6 shows the clock cycle gain that is achieved during Matrix-Vector multiplication of size $(48\times128)\times(128\times1)$ using the SIMD and pipelining combination over a Single Instruction Single Data (SISD) architecture.

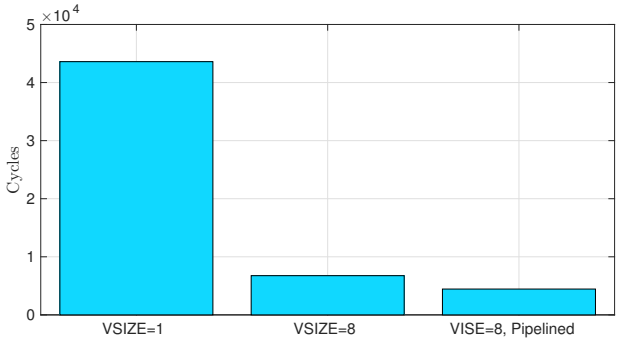
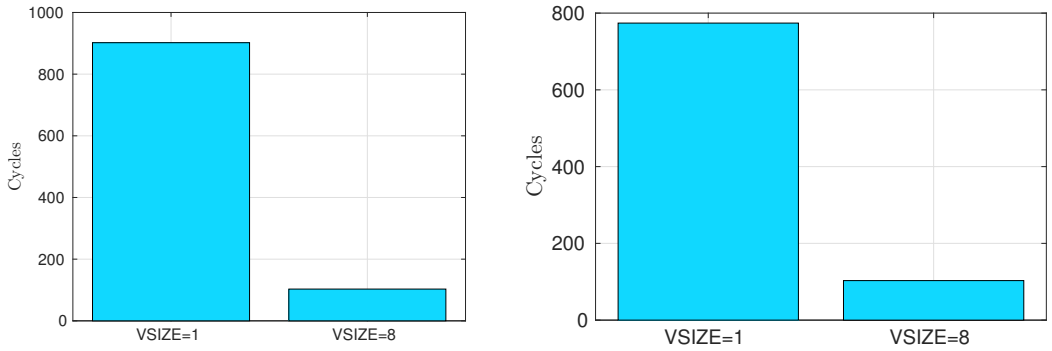


Figure 6.6: Cycle gain on Matrix-Vector products. Vectorizing the operation gives an almost linear reduction, while pipelining and combining a set of instructions in one leads to some extra reduction in cycles

6.3.2 Vector-Vector multiplications and additions

The vectorized architecture of section 6.3.1 can perform adequately well on the tasks of element-wise additions and multiplications of vectors. The gain on cycles is



(a) Vector-Vector multiplications

(b) Vector-Vector additions

Figure 6.7: Cycle gain in Vector-Vector operations. The gain in vector vector operations is almost linearly related to the SIMD size.

proportional to VSIZE, as seen in Figure 6.7.

6.3.3 Non linear activation functions

In section 5.5 it was shown that table-lookup implementations that coarsely approximate the nonlinear $\sigma(x)$ and $\tanh(x)$ functions induce no accuracy loss in the LSTM/FC machine learning architecture. The gains from this observation are realized in two ways.

Firstly, the SIMD vector unit is used to evaluate VSIZE activation functions in parallel, since there is no data dependencies between the various network outputs.

Secondly, in the algorithmic level, the activation functions are implemented with LUTs, and therefore require a vastly smaller number of cycles than a Taylor approximation or Cody-Waite polynomial approximation.

6.3.4 Custom word length

The effect of short word lengths on the application's accuracy were thoroughly investigated in Chapter 5. In the design of the ASIP, the reduction of the word length does not impact the amount of cycles needed for the computations nor the number of computations themselves. However, it directly impacts the processor's energy consumption. This will be further analysed in the energy model described in section 6.4.

6.3.5 Computation skipping

During the ASIP design, computation skipping is implemented through direct hardware checks. These hardware checks are implemented on the ALU of the ASIP, and avoid the switching of its multipliers if an operand is zero. Instead, a zero is directly forwarded on the output. The primitive function that describes the hardware check and the generated Verilog code that implements it are shown in appendix A.3.2.

This approach does not impact the overall number of cycles of any computation, but realizes energy savings by reducing the overall switching activity of the processor. This will be further analysed in the energy model described in section 6.4.

6.4 The Energy model

6.4.1 Introduction

In any digital design methodology, propagation delay, energy and power are terms that arise very often and significantly influence decisions. In general, delay and energy are inversely proportional measures that, along with many other design parameters, create a very interesting and rich optimization space. The optimal point in the energy-delay domain differs for each application. Therefore, in order to better navigate this domain, it is important to understand how power is dissipated in digital circuits and what design parameters affect system performance in terms of delay, since each application can have its own temporal constraints.

6.4.2 Power Dissipation in CMOS circuits

In CMOS technologies, power dissipation can be classified as dynamic or static. Dynamic power dissipation is dependent on the switching activity of the digital circuit while static power refers to the power dissipated from non-switching or switching transistors due to technology imperfections. In the past, static power dissipation (mainly due to transistor leakage currents) was of limited importance, but as feature size shrinks and leakage current becomes significant with respect to operational current, static power dissipation should be treated as important and handled with care.

The total power dissipated in a CMOS circuit can be modelled as:

$$P = P_{dyn} + P_{stat} \tag{6.3}$$

The dynamic and static power dissipation can be modelled as:

$$P_{dyn} = \alpha(C_L + C_{SC}){V_{DD}}^2 f \tag{6.4}$$

$$P_{stat} = (I_{DC} + I_{Leak})V_{DD} \tag{6.5}$$

where:

α = Switching activity factor
C_L = Load capacitance
C_{SC} = Short circuit capacitance
V_{DD} = Supply voltage
f = Clock frequency
I_{DC} = Static current
I_{Leak} = Leakage current.

The switching activity factor α influences dynamic power dissipation only and depends on the switching probabilities of all the transistors in a CMOS circuit. Since this number can easily be in the millions or billions, it is important to observe two points:

- The prediction (and therefore modelling) of the dynamic power dissipation is a very difficult task because it requires the knowledge of the switching activity α, which non-trivially depends on the switching probabilities of all the transistors in a design
- Dynamic power dissipation is data dependent, since switching depends on operations and operations depend on data.

To summarize, power dissipation in digital circuits has two major components, called dynamic and static dissipation. In a functioning digital circuit, the former component is proportional to the switching activity that relies on the operation at hand, while the latter is considered stable throughout the time the circuit operates. This can be summed up in the following equation:

$$P = \alpha(C_L + C_{SC}){V_{DD}}^2 f + (I_{DC} + I_{Leak})V_{DD} \tag{6.6}$$

or, more descriptively,

$$P = \frac{\text{energy}}{\text{operation}} \times \text{rate} + \text{static power} \tag{6.7}$$

A key observation that arises from equation 6.7 is that a given operation (e.g. a multiplication) can be characterised by its energy cost. Therefore, the dynamic power dissipation is a function of a) the operations that need to be performed and b) the switching activity. On the other hand, static dissipation is better captured as a power quantity.

6.4.3 Energy savings through SIMD

In CMOS processes propagation delay is inversely proportional to supply voltage, as is shown in Figure 6.8. Therefore, by allowing for longer propagation delays t_p it is possible for a design to operate at a smaller supply voltage V_{DD} and therefore exploit the quadratic effect that V_{DD} has on dynamic power dissipation to drastically lower energy consumption.

One method of allowing for longer t_p is the usage of concurrency. Creating a hardware parallel implementation of a computational module can have significant benefits. Figure 6.9 shows a reference implementation and Figure 6.10 a parallel implementation of a computational module. Since the two branches of the parallel implementation can work in tandem, they can operate in half the required frequency $f_{par} = \frac{f_{ref}}{2}$ and therefore allow for delay $t_p\prime = 2t_p$. Depending on the technology, this allows for a reduction of the supply voltage by a factor ϵ_{par} which in turn affects dynamic power dissipation:

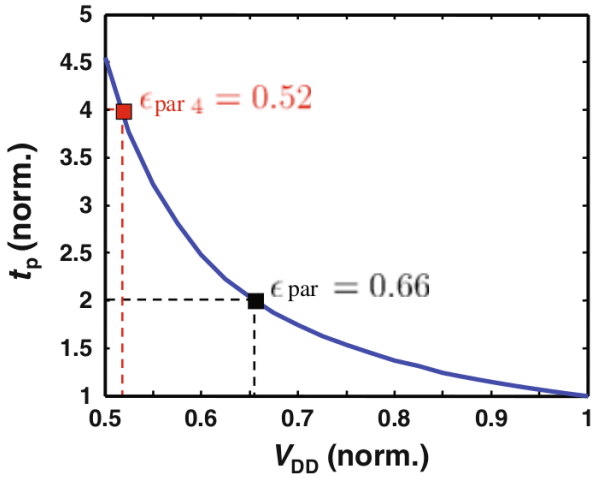


Figure 6.8: Dynamic Voltage Scaling. By allowing for more delay t_p by implementing pipelining or paralellization we can reduce supply voltage V_{DD} and gain quadratic energy savings.

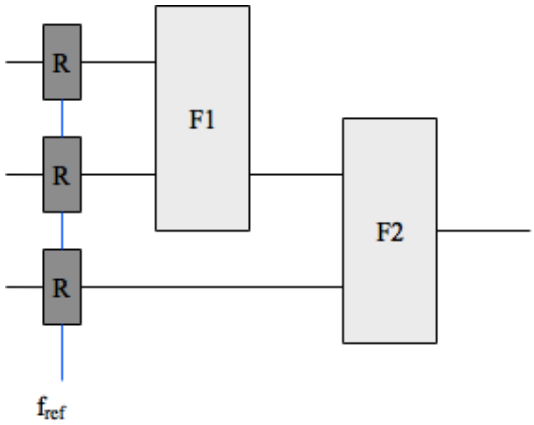


Figure 6.9: A reference implementation of a computational module.

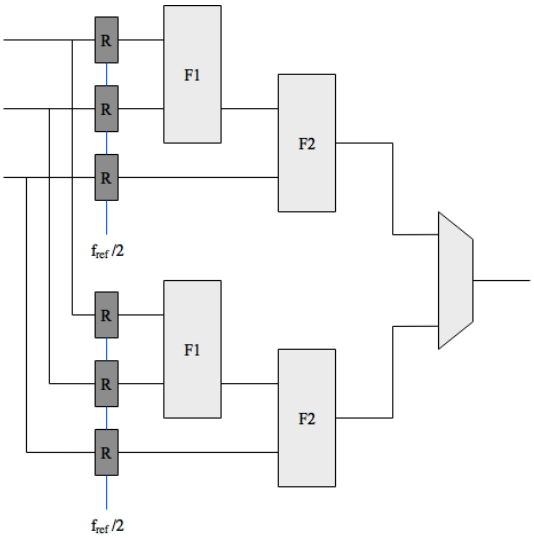


Figure 6.10: A parallel implementation. By replicating the datapath, more computations are achieved in a clock cycle and therefore frequency and voltage scaling can be applied. The tradeoffs involve more chip area and an increase in static power dissipation.

$$f_{par} = \frac{f_{ref}}{2} \tag{6.8}$$

$$C_{par} = (2 + ov_{par})C_{ref} \tag{6.9}$$

$$V_{DDpar} = \frac{\epsilon_{par}}{V_{DDref}} \tag{6.10}$$

$$P_{par} = (\epsilon_{par})^2 \left(\frac{2 + ov_{par}}{2} \right) P_{ref} \tag{6.11}$$

Since this reduction has a quadratic effect, architectures that exploit parallelism, such as the SIMD-enabled ASIP under design, allow for big margins in energy efficiency.

In the parallel implementation of Figure 6.10 a multiplexer operating at f_{ref} is needed to recombine the outputs of the functional units into a stream. This necessary multiplexing overhead is considered small in the typical case and it corresponds to a slight increase in switching capacitance, modelled in the equations by the factor ov_{par}. Also, heavily parallelizing designs leads to a substantially larger area footprint, since the data path of the functional units needs to be replicated and multiplexed. This will also amount to a directly proportional increase in static power dissipation.

As it can be seen in Figure 6.8 the inverse proportionality between V_{DD} and t_d suggests that after a point, driving V_{DD} further down leads to a big leap upwards in t_d, which will require an even more parallel architecture to meet the delay restrictions.

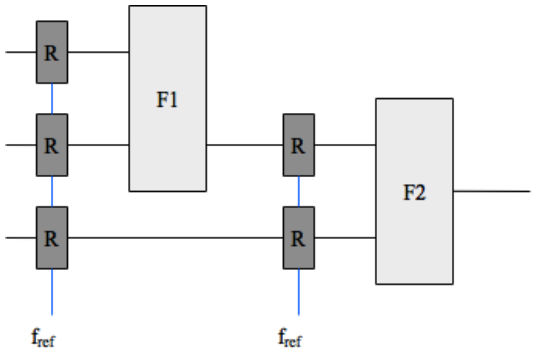


Figure 6.11: A pipelined implementation. Shortening the critical path length by introducing pipeline registers allows for the relaxation of timing constraints with a small capacitance overhead due to the extra registers.

Hence, there exists an optimum amount of concurrency that can be implemented to achieve maximum energy efficiency.

6.4.4 Energy savings through pipelining

By introducing pipeline registers as shown in Figure 6.11 the critical path of the design is shortened and therefore the requirements on t_d can be relaxed. In this way, V_{DD} can be scaled by a factor of ϵ_{pipe} which will again influence dynamic power consumption in a quadratic fashion.

$$f_{pipe} = f_{ref} \tag{6.12}$$
$$C_{pipe} = (1 + ov_{pipe})C_{ref} \tag{6.13}$$
$$V_{DDpipe} = \epsilon_{pipe} V_{DDref} \tag{6.14}$$
$$P_{pipe} = (\epsilon_{pipe})^2(1 + ov_{pipe})P_{ref} \tag{6.15}$$

This approach improves throughput at the expense of latency and also introduces an increase in switching capacitance modelled as ov_{pipe}, due to the switching of the intermediate pipeline registers. Pipelined architectures can have similar power savings with parallel designs although with a smaller area footprint, since the only additions in the design are these intermediate registers. Furthermore, these approaches can be combined to drive V_{DD} further down.

In the ASIP under design, energy savings are realized by defining multiple pipeline stages and dividing microarchitecture instructions in stages. This allows for concurrent execution of instructions, lowers the overall cycles needed and therefore relaxing time delay constraints. The nML code that implements the custom instructions can be found in the appendix A.3.1

	×	+
$\mathbf{f_t}$	$hd + h^2 + \sigma_1 h$	$hd + h^2 + 2h + \sigma_2 h$
$\mathbf{o_t}$	$hd + h^2 + \sigma_1 h$	$hd + h^2 + 2h + \sigma_2 h$
$\mathbf{i_t}$	$hd + h^2 + \sigma_1 h$	$hd + h^2 + 2h + \sigma_2 h$
$\tilde{\mathbf{C}}_\mathbf{t}$	$hd + h^2 + t_1 h$	$hd + h^2 + 2h + t_2 h$
$\mathbf{C_t}$	$2h$	h
$\mathbf{h_t}$	$h + t_1 h$	$t_2 h$
Total	$4(hd + h^2) + 3\sigma_1 h + 2t_1 h + 3h$	$4(hd + h^2 + 2h) + 3\sigma_2 h + 2t_2 h + h$
	$87,528$	$87,936$

Table 6.2: Multiplications and additions in the LSTM forward pass. For a fixed input size d ($d = 39$ in the LSTM/FC architecture) the total number of operations in dominated by the network's size h ($h = 128$ in the LSTM/FC).

6.4.5 Energy savings through smaller word lengths

Using smaller word lengths does not have an effect on the total number of cycles that the processor needs to complete on LSTM/FC forward pass; However, the switching activity α from equation 6.4 depends on the number of operations needed and on the word length that is used for these operations. This is shown in Table 6.1.

	×	+	Reg.	Wire	SRAM
α	$\mathcal{O}(n^2)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(n_{max})$

Table 6.1: The relation between switching activity and elementary operations

In order to estimate the effect of a change in word length on energy consumption, it is necessary to know how many elementary operations (and of which kind) are needed for the LSTM/FC architecture. To avoid confusion and excessive notation, the following analysis is done for one LSTM block. The second block in the design is considered identical, and the analysis is exactly the same for the fully connected layer.

As explained in Section 4.2 and shown in Algorithm 1, the LSTM forward pass consists of the calculation of the outputs of its four gates ($\mathbf{f_t}$, $\mathbf{i_t}$, $\mathbf{o_t}$, $\tilde{\mathbf{C}}_\mathbf{t}$), the update of its state $\mathbf{C_t}$ and the computation of its output $\mathbf{h_t}$.

The total number of multiplications M and the additions A of the LSTM forward pass can be found in Table 6.2:

$$M = 4(hd + h^2) + 3\sigma_1 h + 2t_1 h + 3h \tag{6.16}$$

$$A = 4(hd + h^2 + 2h) + 3\sigma_2 h + 2t_2 h + h \tag{6.17}$$

The switching activity α depends on both M and A:

$$\alpha \propto \mathcal{O}(Mn^2) \tag{6.18}$$
$$\alpha \propto \mathcal{O}(An) \tag{6.19}$$
$$\alpha \propto \mathcal{O}(Mn^2) + \mathcal{O}(An) \tag{6.20}$$

The energy spent in the LSTM forward pass is proportional to the dynamic power, and therefore also depends on the switching activity and the number of operations:

$$E(n) \propto P_{dyn}(n) \propto \alpha \tag{6.21}$$

For a given technology, the energy cost per operation can be modelled as a function of the architecture and the word length:

$$P_{mul}(n) = c_m * n^2 \tag{6.22}$$
$$P_{add}(n) = c_a * n \tag{6.23}$$

From simulations we did in a 40nm Low Power CMOS technology operating at 1.1V simulating 8-bit adder and an 8-bit multiplier, we can approximate c_m and c_a as:

$$c_m = 5.503 \tag{6.24}$$
$$c_a = 2.825 \tag{6.25}$$

This leads to an expression of energy spent in the LSTM forward pass with respect to the number of multiplications M, additions A, relative cost of operations c_m and c_a and word length n:

$$E(n) = \left[\frac{M}{M+A}c_m n^2 + \frac{A}{M+A}c_a n\right]\left[M + A\right] \tag{6.26}$$

The reason that $[M + A]$ is factored outside equation 6.26 is because it represents the total number of operations in the forward pass; When investigating the effect of word length on energy, the total number of operations remains unchanged. This is in contrast with the analysis of the following section, where savings in energy will be realized by reducing the amount of actual computations needed.

According to equation 6.26, equation 6.27 defines energy gain that can be obtained when reducing word length from n_1 to n_2:

$$E_{gain}(n_1, n_2) = \frac{E(n_1)}{E(n_2)} = \frac{Mc_m {n_1}^2 + Ac_a n_1}{Mc_m {n_2}^2 + Ac_a n_2} \tag{6.27}$$

The analysis in Chapter 5 showed that even 8 bits of precision are enough for no performance loss. Therefore, using 16-bit fixed-point arithmetic as a baseline for energy and substituting M, A, c_m, c_a n_1 and n_2 we can achieve an energy savings up to:

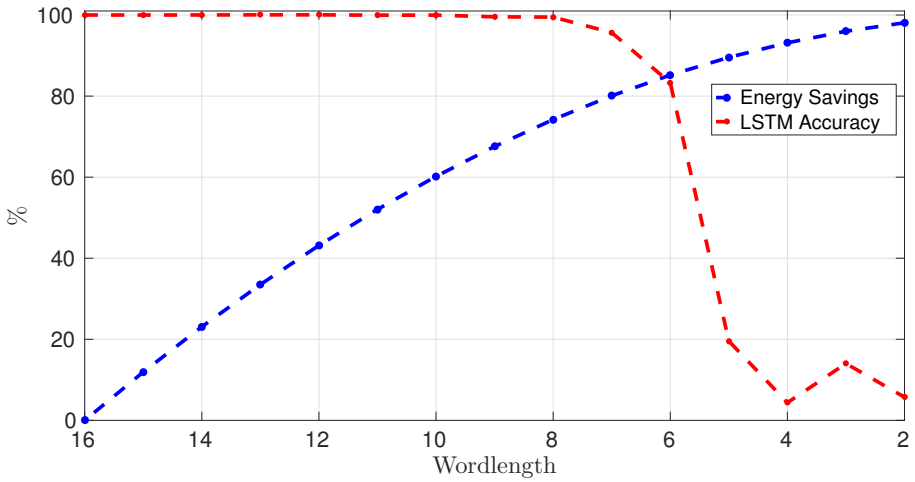


Figure 6.12: Energy savings through shorter word lengths. This figure illustrates how reducing the word length can lead to disproportionate reductions in energy consumption over accuracy. This phenomenon can be exploited with an application-tailored hardware architecture, e.g. an ASIP.

$$E_{gain}(16,8) = \frac{E(16)}{E(8)} = 3.82\times \tag{6.28}$$

$$E_{gain}(16,8) = \frac{E(16)}{E(7)} = 5.02\times \tag{6.29}$$

which amounts 74% energy savings over the baseline with no loss in prediction performance and to 80% with 4% loss in relative accuracy. Figure 6.12 overlays the relative energy reduction and the relative accuracy loss over the word length used in the computations of the LSTM/FC forward pass.

6.4.6 Energy savings through computation skipping

In section 5.4 it was reasoned that since the parameters of the LSTM are known beforehand, it would make sense to prune unnecessary connections by zeroing parameters under a specific threshold. Subsequently, it was mentioned that by preventing circuitry from switching when facing an operation with known result (such as a multiplication with zero) we could decrease the switching activity of the circuit and therefore save energy. This approach is modelled below.

Let s be a factor that models the amount of pruned connections and equivalently the sparsity of the parameter matrices. Let m be the fraction of the overall multiplications M that is due to matrix-vector products where the matrix is known beforehand.

Based on equations 6.26 and 5.7, the energy consumed in the forward LSTM pass as a function of the sparsity factor s can be expressed as:

$$E(s) = \Big[(1-s)mM + (1-m)M\Big] * c_m n^2 + Ac_a n \tag{6.30}$$

In equation 6.30, the first term in the square brackets represents the number of multiplications that are skipped, while the second term the amount of multiplications that originate in other points in the forward pass and cannot be efficiently skipped.

Using data from Table 6.2 we can see that m can be expressed with respect to the input size d hidden size h, and parameters σ_1 and t_1 as follows:

$$m = \frac{4(hd + h^2)}{4(hd + h^2) + 3\sigma_1 h + 2t_1 h + 3h} \tag{6.31}$$

Assuming $d = 39$, $h = 128$ and $\sigma_1 = t_1 = 1$:

$$m = 0.988 \tag{6.32}$$

This effectively means that for LSTM network under examination, 98.8% of the multiplications originate from matrix-vector products with known matrices and can therefore be skipped with predictable results.

In section 5.4 the amount of connection pruning that an LSTM can tolerate was investigated, and it was shown that with 40% of pruning there is no penalty in accuracy, while with 70% of connections pruned, relative accuracy remains above 96%.

In a similar way with equation 6.27 we can define the energy savings factor as a function of the sparsity factor s as the ratio of the baseline energy of a 16-bit fixed-point unpruned forward pass $E(0)$ and the energy of a 16-bit fixed-point pruned pass, $E(s)$:

$$E_{gain}(s) = \frac{E(0)}{E(s)} = \frac{Mc_m n^2 + Ac_a n}{\Big[(1-s)mM + (1-m)M\Big] * c_m n^2 + Ac_a n} \tag{6.33}$$

Assuming sparsity factors $s = 0.4$ and $s = 0.7$, we have, respectively:

$$E_{gain}(0.4) = \frac{E(0)}{E(0.4)} = 1.62\times \tag{6.34}$$

$$E_{gain}(0.7) = \frac{E(0)}{E(0.7)} = 3.02\times \tag{6.35}$$

which amounts to 38% and 67% energy reduction over the baseline, with 0% and 4% relative accuracy loss respectively. Figure 6.13 overlays the relative energy reduction and the relative accuracy loss over the percentage of pruned connections.

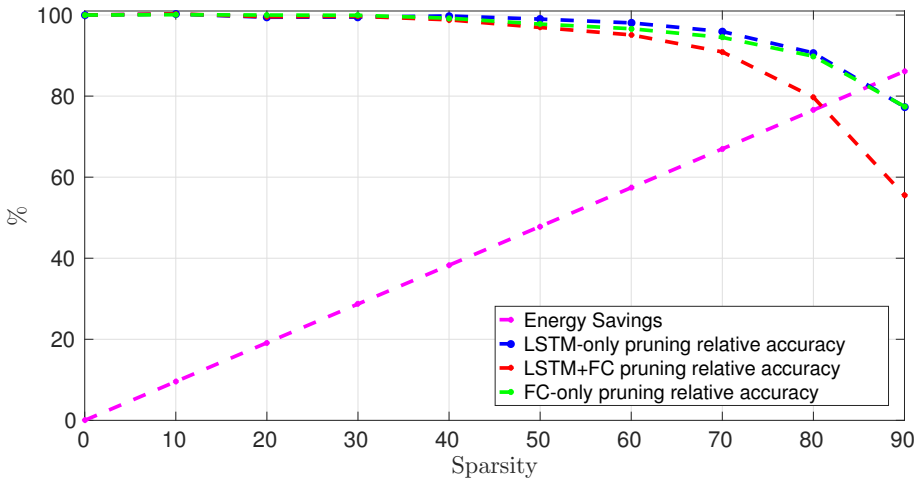


Figure 6.13: Energy savings through connection pruning. This figure illustrates how connection pruning can be exploited in LSTMs to reduce energy consumption without sacrificing accuracy.

6.5 Conclusion

Application Specific Instruction-set Processors bridge the gap between high energy efficiency offered by ASICs and flexibility and programmability offered by CPUs and GPUs. The ASIP design flow allows for optimizations in many abstraction levels allowing for energy-efficient solutions with no sacrifice in flexibility.

In this Chapter, the LSTM/FC architecture first introduced in Chapter 4 is examined from a hardware point of view. The operations needed for the forward pass and the access patterns are investigated. Drawing from 5, specific modifications are made to an ASIP in order to optimize, from an energy standpoint, the forward pass of the LSTM/FC architecture on the task of Automatic Speech Recognition. These optimizations include vector processing, introduction of new pipeline stages, augmentation of the Instruction Set Architecture and RTL modifications to allow for hardware checking that simplifies the high level implementation.

Energy savings up to 80% can be achieved by using shorter word lengths in the data path, while switching activity can be further reduced by 67% more by a combination of connection pruning in the neural network and hardware-driven computation skipping. The pipelining and SIMD instructions introduced can relax timing constraints, allow for Frequency and Voltage Scaling for further energy savings.

Chapter 7

Conclusion

This work addressed the performance-energy tradeoffs that can be achieved in Long Short-Term Memory neural architectures. LSTM neural networks are considered state-of-the-art solutions for a variety of sequential machine learning tasks not limited to Automatic Speech Recognition, but also including video and action classification, unsegmented handwriting recognition and generation, polyphonic music modelling and protein secondary structure prediction.

In Chapters 3 and 4 an architecture that incorporates two LSTM blocks and one fully connected layer, called LSTM/FC, was implemented in the Torch machine learning framework and trained over the TIMIT speech corpus, which consists of spoken English sentences annotated to the phone level. The Framewise Phone Classification Error (FCE) was used as a metric of performance, and the best performing network achieved an FCE of 29.8%. This corresponds to 70.2% accuracy in phone prediction. The architecture showed remarkable ability to learn, achieving peak performance significantly close to the state of the art in much less time than other, non-recurrent neural architectures.

In Chapter 5 an analysis on the error resilience of LSTM networks was performed, since the ultimate goal was the application of approximate computing techniques that can lead to significant energy savings by allowing for small computational error, instead of perfectly accurate results. Similar analyses had been documented for fully connected neural networks and other architectures like convolutional neural networks, but an analysis of this kind for LSTM networks had not yet been performed.

The investigation followed three directions: a) The effect of quantization of LSTM parameters and data was addressed, and it proved that LSTMs suffer no accuracy loss even when using 8 bits for their parameters and calculations b) The effect of connection pruning was addressed, and it proved that no accuracy loss occurs even with 40% of their connections pruned, while allowing for a 4% relative accuracy reduction allows for 70% pruning and c) The non-linear computationally expensive activation functions were replaced with easy to implement look-up table approximations. It is shown that the easiest to implement interpolation mechanism that uses a power-of-2 breakpoint achieves perfect relative accuracy using a look-up table with no more than 8 entries in the worst case.

In Chapter 6 the LSTM/FC architecture was examined from a hardware point of view and the operations needed for the forward pass were investigated. Specific modifications were made to an Application Specific Instruction-set Processor in order to optimize, from an energy standpoint, the forward pass of the LSTM/FC architecture on the task of Automatic Speech Recognition. These optimizations include vector processing, the introduction of new pipeline stages, the augmentation of the Instruction Set Architecture and RTL modifications to allow for hardware checking that simplifies the high level implementation. Energy savings up to 80% were achieved by using shorter word lengths in the data path, while switching activity was further reduced by 67% by a combination of connection pruning in the neural network and hardware-driven computation skipping. The pipelining and SIMD instructions that were introduced relaxed timing constraints and allowed for Frequency and Voltage Scaling for further energy savings.

Using approximate computing methods for energy savings is hereby proven possible on LSTM neural networks. The dramatic energy savings that can be achieved with an educated choice of architectural and algorithmic modifications can enable LSTM-based machine learning on platforms with very tight power budgets. Applications where LSTM-based learning can prove useful include computer vision, action classification, systematic genomic identification, text understanding, user experience design, forecasting, fraud or failure detection etc., while such platforms include wireless sensor networks, mobile phones, ubiquitous and wearable electronics, aerial unmanned vehicles, medical implants, personal electronics, and many other platforms operating in non-energy-rich environments.

Appendices

Appendix A

Code

A.1 LSTM modelling, training and testing in Torch

In this section, the most important parts of Lua/Torch code written are presented. The code in this section deals with:

- The import of the TIMIT dataset in Torch and it's manipulation into a usable format
- The implementation of the LSTM/FC architecture presented in Chapter 4
- The training of the architecture on the training set of TIMIT
- The benchmarking of the trained system on the TIMIT validation set.

A.1.1 Data read from HDF5 files into Torch tensors

```
1  --[[
2  This script uses the hdf5 library to load tables from HDF5 files into
       Torch Tensors.
3  --]]
4  require 'torch'
5  require 'hdf5'
6  local f = hdf5.open('timit_reduced_mfcc.h5', 'r')
7  --training set load
8
9  training_targets=           f:read('/reduced/training/targets'):all()
10 training_masks=             f:read('/reduced/training/masks'):all()
11 training_labels_reduced=    f:read('/reduced/training/labels_reduced'):
       all()
12 training_labels=            f:read('/reduced/training/labels'):all()
13 training_feats=             f:read('/reduced/training/default'):all()
14
15 --validation set load
16
17 validation_targets=           f:read('/reduced/validation/targets'):all
       ()
18 validation_masks=             f:read('/reduced/validation/masks'):all()
```

```
validation_labels_reduced=    f:read('/reduced/validation/
    labels_reduced'):all()
validation_labels=            f:read('/reduced/validation/labels'):all
    ()
validation_feats=             f:read('/reduced/validation/default'):all
    ()

--test set load

test_targets=           f:read('/reduced/test/targets'):all()
test_masks=             f:read('/reduced/test/masks'):all()
test_labels_reduced=    f:read('/reduced/test/labels_reduced'):all()
test_labels=            f:read('/reduced/test/labels'):all()
test_feats=             f:read('/reduced/test/default'):all()

f:close()
```

A.1.2 Network implementation and training

```
--[[
This script uses the rnn library from https://github.com/Element-
    Research/rnn to implement a Bidirectional LSTM architecture
    consisting of two parallel LSTM layers and a fully connected layer.
It creates the forward LSTM as a Torch module, clones it and passes it
    to the BiSequencer decorator that will pass the same sequence both
    forward and backward to the two LSTMs. The whole architecture is
    wrapped in a rnn.Sequential module that can be trained with full
    BPTT by unrolling during training. The cross entropy criterion is
    used, wrapped into a rnn.SequencerCriterion that makes it applicable
     for training on sequences.
--]]


require 'rnn'
require 'optim'
--Set up logging
local date=os.date('%Y_%m_%d_%X')
local exp_id = string.format('%s_%s', 'exfceiment', date)
local log_id= exp_id .. '.log'
logger=optim.Logger(log_id)

--Import data from HDF5 file into Torch Tensors
dofile('importTIMIT.lua')
--Load the feature sequences
feats=training_feats:fcemute(2,1,3)
--Load the associated labels
labels=training_targets:fcemute(2,1,3)
--Add 1 to the labels (ranging in [0-60]) to match lua table indexing
labels=labels+1
--Load the masks
masks=training_masks:fcemute(2,1,3)


--Set network size
local inputSize = 39
```

```
local hiddenSize = 128
local outputSize = 61
--Set minibatch size. Here it is set to 1: Parameters are updated after each sequence.
local batchSize=1

--Extract the size of the training set
local dsSize=feats:size(1)
local seqLength=feats:size(2)


--Generate checkpoints of the model after every epoch
function save(model, exp_id, numEpochs)
   local filename = exp_id .. '.model.' .. numEpochs
   print('Saving model as ' .. filename)
   torch.save(filename, model)
end

--Load saved data or reshape the new
function build_data()
    local inputs={}
    local targets={}
    if use_saved then
      print('Using saved data...')
      inputs = torch.load('inputs.t7')
      targets = torch.load('targets.t7')
   else
    for i=1, dsSize do
      local input=feats[i]:narrow(1,1,seqLength)
      local target=labels[i]:narrow(1,1,seqLength)

      table.insert(inputs,input)
      table.insert(targets,target)
    end
   end
   return inputs, targets
end

--Build the network architecture
function build_network(inputSize, hiddenSize, outputSize)
  if use_saved then
      print('Using saved model...')
      rnn = torch.load('trained-model.t7')
  else
  --BLSTM implementation
      fwd = nn.FastLSTM(inputSize, hiddenSize)
      blstm=nn.BiSequencer(fwd) --BiSequencer creates a clone of fwd to be trained backwards
      rnn = nn.Sequential() --Sequential wraps the blstm and the fully connected layer
      :add(blstm)
      :add(nn.Sequencer(nn.Linear(hiddenSize*2,outputSize))) --hiddenSize*2 due to BLSTM

   --[[
```

```
80    --GRU implementation
81        rnn = nn.Sequential()
82        :add(nn.GRU(inputSize, hiddenSize))
83        :add(nn.Linear(hiddenSize,outputSize))
84        rnn=nn.Sequencer(rnn)
85    --]]
86    end
87    return rnn
88 end
89
90
91 -- Main function
92 --The data is built.
93 inputs, targets = build_data()
94
95 --The network is built.
96 rnn = build_network(inputSize, hiddenSize, outputSize)
97
98 --The architecture is output to a text file for logging.
99 logger:add{['model architecture'] = tostring(rnn)}
100
101 --The training criterion is defined and wrapped in a SequencerCriterion
       to work on sequences.
102 crit=nn.CrossEntropyCriterion()
103 seqC = nn.SequencerCriterion(crit)
104
105 --The network's parameters are cloned to a "light" model that doesn't
      keep the internal state and is very small in size so it can be
      efficiently saved and restored.
106 parameters, gradParameters = rnn:getParameters()
107 light_rnn=rnn:clone('weight','bias','running_mean','running_std')
108
109 --The initial learning rate is set.
110 local lr=0.005
111
112 --The network is initialized for training.
113 rnn:training()
114
115 --Training begins
116 for numEpochs=0,100 do
117   logger:setNames{'Epoch', 'Sequence', 'Loss'}
118   print('Epoch ',numEpochs)
119
120   local err = 0
121   local err_sum=0
122   local start = torch.tic()
123
124   print('Learning Rate: ',lr)
125    for seqNum=1,dsSize do
126
127         --Each sequence is prepared to be of its original size.
128         inputSequence  = {}
129         expectedTarget = {}
130         mask=masks[seqNum]
131         seqLength=mask[mask:eq(1)]:size()[1]
```

```
132         for j=1,seqLength do
133          table.insert(inputSequence, inputs[seqNum][j])
134          expectedTarget[j]=targets[seqNum][j]
135         end
136
137
138         rnn:zeroGradParameters()
139     --The forward pass of the sequence is fceformed.
140         out = rnn:forward(inputSequence)
141         --The error based on the network's output and the expected
      output is calculated.
142         err = seqC:forward(out, expectedTarget)
143         err_sum=err_sum+err
144
145     --The backpropagation algorithm is executed.
146         gradOut = seqC:backward(out, expectedTarget)
147         rnn:backward(inputSequence, gradOut)
148
149         --The network's parameters are updated with the specified
      learning rate.
150         rnn:updateParameters(lr)
151
152         print ('Epoch #', numEpochs .. 'Utterance #', seqNum .. ' of ',
       dsSize .. '. Loss: ', err)
153         logger:add{numEpochs,seqNum,err}
154
155     end
156
157     local currT=torch.toc(start)
158     average_loss=err_sum/dsSize
159     print('Epoch #', numEpochs .. ' average loss', average_loss .. 'in ',
       currT .. ' s')
160     logger:setNames{'Epoch', 'average_loss', 'time','learning_rate'}
161     logger:add{numEpochs, average_loss, currT, lr}
162
163
164     --After each epoch, the model is saved so it can be reloaded if
      necessary.
165     save(light_rnn,exp_id,numEpochs)
166
167     --The model is benchmarked on the validation set.
168     print('Model is now tested on the validation set...')
169     dofile('benchmark_validation.lua')
170     --The learning rate is adjusted in a "bold driver" fashion. It is
      slowly incremented if the loss decreases and sharply decremented if
      the loss increases over the epochs.
171     if numEpochs~=0 then
172       if average_loss>=previous_average_loss then
173         lr=lr/2
174       else
175         lr=lr*1.01
176       end
177     else
178       previous_average_loss=average_loss
179     end
```

```
previous_average_loss=average_loss
end
```

A.1.3 Network benchmarking on the TIMIT validation set

```
--[[
This script evaluates a preloaded model on data imported from the TIMIT
     dataset. It loads the data of the validation set, and performs the
    forward pass over all the sequences in the set. The Framewise
    Classification Error (FCE) is calculated for each sequence and the
    average FCE is also calculated over the validation set.
--]]

require 'rnn'
require 'optim'
--The validation set is loaded in Torch Tensors.
dofile('importTIMIT.lua')
local feats=validation_feats:fcemute(2,1,3)
local labels=validation_targets:fcemute(2,1,3)
labels=labels+1
local masks=validation_masks:fcemute(2,1,3)

--The size is extracted.
local dsSize=feats:size(1)
local maxSeqLength=feats:size(2)

function build_data()
    local inputs={}
    local targets={}
    if use_saved then
      print('Using saved data...')
      inputs = torch.load('training.t7')
      targets = torch.load('targets.t7')
   else
    for i=1, dsSize do
      input=feats[i]:narrow(1,1,maxSeqLength)
      target=labels[i]:narrow(1,1,maxSeqLength)
      table.insert(inputs,input)
      table.insert(targets,target)
    end
   end
   return inputs, targets
end


--MAIN

--Data are built.
local inputs, targets = build_data()
local fce_acc=0
logger:setNames{'Benchmarking on the validation set'}
logger:setNames{'seqNum', 'FCE'}
--The evaluation begins.

rnn:evaluate()
```

```
for seqNum=1,dsSize do

        local inputSequence = {}
        local expectedTarget = {}
        local max={}
        local index={}

        local mask=masks[seqNum]
        local seqLength=mask:eq(1):size()[1]
        for j=1,seqLength do
         table.insert(inputSequence, inputs[seqNum][j])
         expectedTarget[j]=targets[seqNum][j]
        end

  --The network's state is reset between sequences.
        rnn:zeroGradParameters()
  --A sequence is forwarded
        local out = rnn:forward(inputSequence)
  --A sequence of predicted labels is extracted from the output of the
    network, which is label probabilities.
        local predictedTarget={}
        for i=1,seqLength do
          max[i], predictedTarget[i] = torch.max(out[i],1)
        end
        local predictedTensor= torch.Tensor(seqLength)
        local expectedTensor=torch.Tensor(seqLength)
        local sum=0

  --Calculate the Framewise Classification Error
        for i=1,seqLength do
          predictedTensor[i]=predictedTarget[i][1]
          expectedTensor[i]=expectedTarget[i][1]


          if predictedTensor[i]==expectedTensor[i] then
            sum=sum+1
          end
        end
        local fce=(1-sum/(seqLength))*100
        fce_acc=fce_acc+fce
        print('FCE:' .. fce)
        logger:add{seqNum, fce}

  end

--Calculate the average FCE over the validation set.
average_fce=fce_acc/dsSize
print('Average FCE:' .. average_fce)
logger:setNames{'Epoch', 'Average_FCE_val'}
logger:add{counter, average_fce}
logger:setNames{'==========='}
```

A.2 Approximate computing in MATLAB

In this section, the most important parts of the code that simulates the various approximate computing experiments are presented. The code in this section deals with:

- The import of the trained system's parameters into MATLAB, along with the TIMIT validation set
- The implementation of the architecture's forward pass
- The quantization of the system's parameters and data into various wordlengths
- The connection pruning and the encoding of the sparse matrices using Huffman encoding
- The LUT-approximations of the nonlinear $\sigma(x)$ and $\tanh(x)$.

A.2.1 Data import from Torch

```
%% Data import from Torch
load parameters.mat
%w1: Forward LSTM: Forget, input, output and candidate state input-to-hidden weights of size 128*39, for the forward LSTM as one matrix 512*39.
%w2: Forward LSTM: Forget, input, output and candidate state biases of size 128*1, as one matrix 512*1.
%w3: Forward LSTM: Forget, input, output and candidate state hidden-to-hidden weights of size 128*128, as one matrix 512*128.

%w4: Backward LSTM: Forget, input, output and candidate state input-to-hidden weights of size 128*39, as one matrix 512*39.
%w5: Backward Forget, input, output and candidate state biases of size 128*1, as one matrix 512*1.
%w6: Backward Forget, input, output and candidate state hidden-to-hidden weights of size 128*128, as one matrix 512*128.

%w7: Fully Connected: Weights of size 61*256.
%w8: Fully Connected: Biases of size 61*1.

load validation.mat
load seqLengths.mat
%feats: 400 normalized, preprocessed MFCC sequences of size 39*1, padded to the maximum length present in the sequences of the validation set (742). One matrix of size 400*742*39.
%labels: 400 label sequences padded to the maximum length (742). One matrix of size 400*742.
%seqLengths: The length of each of the 400 sequences present in the validation set. One matrix of size 400*1.
```

A.2.2 LSTM Forward pass

```
function [results]= forward_pass()

dsSize=400; %The dataset size
fce_acc=0;

for seqNum=1:dsSize
    %Slice one sequence from the validation set.
    inputSequence=squeeze(feats(seqNum,:,:));
    %And also its target labels.
    expectedSequence=labels(seqNum,1:seqLengths(seqNum));
    seqLength=seqLengths(seqNum);
    defaults= [1  0];
    %For each sequence in the validation set, reset all network states.
    F_i=[];F_f=[];F_z=[];F_c=[];F_o=[];F_h=[];
    B_i=[];B_f=[];B_z=[];B_c=[];B_o=[];B_h=[];
    predictedSequence=[];
    %The first hidden state and output is set to zero.
    c0=zeros(hiddenSize,1);
    h0=zeros(hiddenSize,1);

    %Rename the sequence.
    x=inputSequence';
    x=x(:,1:seqLength);
    %And flip it so that the backward LSTM can process it with the same
    %indexing.
    y=fliplr(x(:,1:seqLength));


    for t=1:seqLength
        %Forward LSTM Calculate all gates with one operation.
        if t==1 %Handle the first frame separately.
            F_all_input_sums = w1*x(:,t)+w3*h0+w2;
        else
            F_all_input_sums = w1*x(:,t)+w3*F_h(:,t-1)+w2;
        end

        %Slice the gates.
        F_n1=F_all_input_sums(1:hiddenSize);
        F_n2=F_all_input_sums(hiddenSize+1:2*hiddenSize);
        F_n3=F_all_input_sums(2*hiddenSize+1:3*hiddenSize);
        F_n4=F_all_input_sums(3*hiddenSize+1:4*hiddenSize);

        %Take the result through the nonlinearities.
        F_in_gate=sigmf(F_n1,defaults);
        F_in_transform=tansig(F_n2);
        F_forget_gate=sigmf(F_n3,defaults);
        F_out_gate=sigmf(F_n4,defaults);
        %Calculate the internal state.
        if  t==1 %Treat first frame separately.
            F_c(:,t)=F_forget_gate.*c0+F_in_gate.*F_in_transform;
        else
            F_c(:,t)=F_forget_gate.*F_c(:,t-1)+F_in_gate.*
    F_in_transform;
        end
        %Calculate the output.
```

```
55          F_h(:,t)=F_out_gate.*tansig(F_c(:,t));
56
57          %Backward The procedure is exactly the same but for the flipped
58          %sequence, y.
59          if t==1
60              B_all_input_sums = w4*y(:,t)+w6*h0+w5;
61          else
62              B_all_input_sums = w4*y(:,t)+w6*B_h(:,t-1)+w5;
63          end
64
65          B_n1=B_all_input_sums(1:hiddenSize);
66          B_n2=B_all_input_sums(hiddenSize+1:2*hiddenSize);
67          B_n3=B_all_input_sums(2*hiddenSize+1:3*hiddenSize);
68          B_n4=B_all_input_sums(3*hiddenSize+1:4*hiddenSize);
69
70          B_in_gate=sigmf(B_n1,defaults);
71          B_in_transform=tansig(B_n2);
72          B_forget_gate=sigmf(B_n3,defaults);
73          B_out_gate=sigmf(B_n4,defaults);
74
75          if t==1
76              B_c(:,t)=B_forget_gate.*c0+B_in_gate.*B_in_transform;
77          else
78              B_c(:,t)=B_forget_gate.*B_c(:,t-1)+B_in_gate.*
    B_in_transform;
79          end
80
81          B_h(:,t)=B_out_gate.*tansig(B_c(:,t));
82
83      end
84      %The backward LSTM output sequence should be flipped so it becomes
85      %aligned with the forward LSTM output sequence.
86      B_h=fliplr(B_h);
87
88      FC_softmax=[];
89      for t=1:seqLength
90          %Fully Connected Layer Concatenate the outputs from the forward
     and
91          %the backward LSTM, and pass them to the fully connected layer.
92          FC_input=cat(1,F_h(:,t),B_h(:,t));
93          FC_output=w7*FC_input+w8;
94
95          %The maximum among the outputs of the Fully Connected layer
96          %specifies the predicted label.
97          [M,I] = max(FC_output);
98
99          %Keep the predicted sequence to compare it against the target
    label
100         %sequence.
101         predictedSequence(t)=I;
102     end
103
104     %Calculate the Framewise Classification Error for the sequence.
105     sum=0;
106     for i=1:seqLength
```

```
107         if predictedSequence(i)==expectedSequence(i)
108             sum=sum+1;
109         end
110     end
111     fce=(1-sum/seqLength)*100;
112     fce_acc=fce_acc+fce;
113     sprintf('FCE: %f',fce);
114     sprintf('pass complete %d',seqNum);
115     fce_list(seqNum)=fce;
116 end
117
118 %Calculate the average FCE over the validation set.
119 average_fce=fce_acc/seqNum;
120 sprintf('Average_fce:%f',average_fce)
121 results=average_fce;
122 end
```

A.2.3 Quantization to different word lengths

```
1  %% Dynamic Fixed Point representation
2  % The parameters and the data are quantized to shorter word lengths to
3  % simulate its effect on network accuracy. The desired wordlength is WL
       .
4  function [w1q w2q w3q w4q w5q w6q w7q w8q featsq] =...
5      quantize(WL,w1,w2,w3,w4,w5,w6,w7,w8,feats)
6
7      %The parameters are concatenated.
8      x=[w1(:) ; w2(:); w3(:); w4(:); w5(:); w6(:); w7(:); w8(:)];
9
10     %The next power of two that is larger than the largest element
11     %of the parameters is calculated.
12     n1=ceil(log2(max(x(:))+1));
13     n2=real(ceil(log2(min(x(:)))));
14
15     nextp2=max(n1,n2);
16
17     %The input space (A,B) is calculated.
18     A=-2^(nextp2);
19     B=2^(nextp2)-1;
20     %The weights will be scaled in (-1,1) before quantization.
21     C=-1;
22     D=1;
23
24     %The data are scaled in (-1,1), multiplied by 2^WL, rounded,
       divided by
25     %2^WL and then rescaled back to their original range.
26     w1q=((round(((w1-A)/(B-A)*(D-C)+C)*2^WL)/2^WL)-C)*(B-A)/(D-C)+A;
27     w2q=((round(((w2-A)/(B-A)*(D-C)+C)*2^WL)/2^WL)-C)*(B-A)/(D-C)+A;
28     w3q=((round(((w3-A)/(B-A)*(D-C)+C)*2^WL)/2^WL)-C)*(B-A)/(D-C)+A;
29     w4q=((round(((w4-A)/(B-A)*(D-C)+C)*2^WL)/2^WL)-C)*(B-A)/(D-C)+A;
30     w5q=((round(((w5-A)/(B-A)*(D-C)+C)*2^WL)/2^WL)-C)*(B-A)/(D-C)+A;
31     w6q=((round(((w6-A)/(B-A)*(D-C)+C)*2^WL)/2^WL)-C)*(B-A)/(D-C)+A;
32     w7q=((round(((w7-A)/(B-A)*(D-C)+C)*2^WL)/2^WL)-C)*(B-A)/(D-C)+A;
33     w8q=((round(((w8-A)/(B-A)*(D-C)+C)*2^WL)/2^WL)-C)*(B-A)/(D-C)+A;
```

```
34
35      %Feature quantization follows the same procedure. It is done
        separately
36      %because it cannot be considered known beforehand, like the
        parameters.
37      x=feats(:);
38      n1=ceil(log2(max(x(:))+1));
39      n2=real(ceil(log2(min(x(:)))));
40
41      nextp2=max(n1,n2);
42
43      A=-2^(nextp2);
44      B=2^(nextp2)-1;
45      C=-1;
46      D=1;
47      featsq=((round(((feats-A)/(B-A)*(D-C)+C)*2^WL)/2^WL)-C)*(B-A)/(D-C)
        +A;
48  end
```

A.2.4 Connection pruning

```
1  %% Connection pruning for a specific sparsity amount.
2  %The percentile that matches the sparsity amount is calculated over the
3  %absolute value of the parameters and all elements with absolute values
4  %lower than that are set to zero.
5  w1(abs(w1)<prctile(abs(w1(:)),sparsity_amount))=0;
6  w2(abs(w2)<prctile(abs(w2(:)),sparsity_amount))=0;
7  w3(abs(w3)<prctile(abs(w3(:)),sparsity_amount))=0;
8  w4(abs(w4)<prctile(abs(w4(:)),sparsity_amount))=0;
9  w5(abs(w5)<prctile(abs(w5(:)),sparsity_amount))=0;
10 w6(abs(w6)<prctile(abs(w6(:)),sparsity_amount))=0;
11 w7(abs(w7)<prctile(abs(w7(:)),sparsity_amount))=0;
12 w8(abs(w8)<prctile(abs(w8(:)),sparsity_amount))=0;
13
14 total_sparsity=(nnz(w1)+nnz(w2)+nnz(w3)+nnz(w4)+nnz(w5)+nnz(w6)...
15     +nnz(w7)+nnz(w8))...
16     /(prod(size(w1))+prod(size(w2))+prod(size(w3))+prod(size(w4))+...
17     prod(size(w5))+prod(size(w6))+prod(size(w7))+prod(size(w8)));
```

A.2.5 Huffman encoding of sparse matrices

```
1  %Huffman encoding of sparse matrices
2  function [results] = huffman_test(x,WL,sparsity_amount)
3      %This function increases the sparsity of x, quantizes to a specific
4      %wordlength and then encodes the matrix by huffman encoding.
5
6      %Increase sparsity
7      x(abs(x)<prctile(abs(x(:)),sparsity_amount))=0;
8
9      %Quantize
10     A=quantize(x,WL)
11     %
12
13     %Create the symbol table
14     C=unique(A);
```

```
15
16      %Create  the a-posteriori probabilities for signals
17      D = zeros(size(C));
18      for i=1:length(C)
19          D(i)=sum(A==C(i))/length(A);
20      end
21
22      %Create the Dictionary
23      [dict avglen]=huffmandict(C,D);
24
25      %Encode the matrix
26      comp=huffmanenco(A,dict);
27      binA=de2bi(A);
28      binComp=de2bi(comp);
29
30      %Calculate the achieved compression
31      compression=numel(binComp)/numel(binA);
32      results = numel(binComp)
33  end
```

A.2.6 Piecewise approximations of nonlinear functions

```
1  %% LUT implementation of the sigmoid simgf(x).
2  function [ y ] = piecewise_sigmf( x,options, mode, xdata,ydata)
3      % x: the input vector
4      % xdata, ydata: LUT Breakpoints and evaluated values of the ideal
5      % sigmf(x).
6      % mode: arguments for the interp1 function. Default: 'linear'
7      % options: dummy argument for the sigmoid.
8
9      space=xdata;
10     values=ydata;
11     %Linear interpolation, with data outsite the domain of interest
12     %extrapolating to 1
13     y=interp1(space,values,x,mode,1);
14     % Extrapolating outside the domain of interest toward negative
       values
15     % should be done by setting the elements to 0.
16     y(x<-8)=0;
17 end
18
19 %LUT implementation of the hyperbolic tangent tansig(x).
20     function [ y ] = piecewise_tansig( x, mode, xdata,ydata )
21     space=xdata;
22     values=ydata;
23     y=interp1(space,values,x,mode,1);
24     y(x<-8)=-1;
25 end
```

A.3 ASIP Description

This section contains the modifications done to the Tvec core by Synopsis.

A.3.1 nML Modifications

```
opn vector_instr(gvec_unit_instr |
                 gvec_vsum_instr |
                 gvec_vinit_instr |
                 gvec_ldst_spidx_instr |
                 gvec_ldst_indir_instr |
                 gvec_vldst_indir_instr |
                 vec_cmp_instr |
                 vc_instr |
     vc_init_instr |
     vec_reduc
                 )
{
    image
        : "100"::gvec_unit_instr
        | "100"::gvec_vsum_instr
        | "100"::gvec_vinit_instr
        | "11"::gvec_ldst_spidx_instr
        | "101"::gvec_ldst_indir_instr
        | "101"::gvec_vldst_indir_instr
        | "000"::"11001"::vec_cmp_instr
        | "001"::"01010"::vc_instr
  | "010"::"010"::vc_init_instr
        | "010"::"011"::vec_reduc
        ;
}

//transitories definition
pipe vec_reduc_pipe<vword>;
trn ag2_addr<addr>;
trn ag2_vaddr<vaddr>;
trn ag1_addr<addr>;
trn vecz<vword>;
trn vecq<vword>;
pipe v0pipe<vword>;
pipe v1pipe<vword>;

//multi-stage pipelined operation for the parallel load of 2 vectors
   and the MAC
opn vec_reduc(ag1: vag_opn)
{
    action {
    stage ID:
    ag1_vaddr = ag1_addr = ag1;
    dm_vaddr = ag1_vaddr;
    ag2_vaddr = ag2_addr = ag2;
    dm2_vaddr = ag2_vaddr;

    stage ID..E1:
        V_0`E1` = vect`E1` = dmv_read`E1` = DMv[dm_vaddr`ID`]`ID`;
        V_1`E1` = vecz`E1` = dmv2_read`E1` = DMv[dm2_vaddr`ID`]`ID`;
    stage E2:
        vecs = V_0;
        vecr = V_1;
        vec_reduc_pipe = vecq = vmul(vecr,vecs)@vec;
    stage E3:
```

```
55          R_0 = rte3 =  vsum(vec_reduc_pipe)@vec;
56      }
57
58      syntax : "load from dm(" ag1 ") to V[0] and dm("ag2") to V[1],
        vec_reduc" ;
59      image  : ag2::ag1;
60  }
61
62  // register declarations
63
64  reg R[8]<word,uint3>              // general purpose registers
65              syntax ("R")
66              read( rrid            // ID stage read port
67                    rre1 rse1 rse2)    // E1 stage read ports
68              write(rtid            // ID stage write port
69                    rte1 rte2 rte3);        // E1 stage write port and E2
        and E3
70  //Aliases to register file
71  reg R_0<word> alias R[0];
72
73  // Aliases to vector register file
74  reg V_0<vword> alias V[0];
75  reg V_1<vword> alias V[1];
```

A.3.2 Primitive Modifications

```
1  task vword_vmul_vword_vword;
2    output signed [127:0] result;
3    reg    signed [127:0] result;
4    input  signed [127:0] a;
5    input  signed [127:0] b;
6    begin : vword_vmul_vword_vword_task
7      reg signed [127:0] rv;
8      integer           i;
9      reg signed [31:0] t;
10     reg signed [15:0] tmp;
11     reg signed [15:0] t_0;
12     reg signed [15:0] t_1;
13     reg signed [31:0] t_2;
14     for (i = 0; i < 8; i = i + 1)
15     begin
16       tmp = $signed(a[{i, 4'b0000}+:16]);
17       if (tmp == 16'sb0000000000000000)
18       begin
19         t = 32'sb00000000000000000000000000000000;
20       end
21       else
22       begin
23         t_0 = $signed(a[{i, 4'b0000}+:16]);
24         t_1 = $signed(b[{i, 4'b0000}+:16]);
25         t_2 = t_0 * t_1;
26         t = t_2;
27       end
28       rv[{i, 4'b0000}+:16] = $signed(t[15:0]);
29     end
```

```
30      result = rv;
31    end
32  endtask
```

A.3.3 C Kernels

```
1  //A SISD implementation of Matrix-Vector product
2  void matrix_vector_sisd(int *A, int *x, int *y, int Arows, int Acols)
3  {
4      int* vA = (int*)A;
5
6      for (int i=0; i<Arows; i++)
7      {
8          *(y+i)=0;
9          vA = (int*)(A+Acols*i);
10
11         for(int j=0;j<Acols;j++)
12             {
13                 *(y+i)+=*(vA+j) * (*(x+j));
14             }
15     }
16 }
17 //A SIMD implementation of Matrix-Vector Product
18 void matrix_vector_simd(int *A, int *x, int *y, int Arows, int Acols)
19 {
20     vint* vA = (vint*)A;
21     vint* vx=  (vint*)x;
22     int reps=Acols/VSIZE;
23     for (int i=0; i<Arows; i++)
24     {
25         *(y+i)=0;
26         vA = (vint*)(A+stride*i);
27         for (int j=0; j<reps; j++)
28         {
29             *(y+i)= *(y+i) + sum( *(vA+j) * (*(vx+j)));
30         }
31     }
32 }
33 //A SISD vector addition
34 void vector_vector_add_sisd(int *x, int *y, int *z, int numel)
35 {
36
37     for (int i=0;i<numel;i++)
38     {
39         *(z+i)= (*(x+i))+ (*(y+i));
40     }
41
42 }
43 //A SIMD vector addition
44 void vector_vector_add_simd(int *x, int *y, int *z, int numel)
45 {
46     vint* vx = (vint*)x;
47     vint* vy = (vint*)y;
48     vint* vz = (vint*)z;
49
```

```
50      int reps=numel/VSIZE;
51
52      for (int i=0;i<reps;i++)
53      {
54          vx=(vint*)(x+VSIZE*i);
55          vy=(vint*)(y+VSIZE*i);
56          vz=(vint*)(z+VSIZE*i);
57          (*vz)=(*vx)+(*vy);
58      }
59  }
60  //A SISD vector multiplication
61  void vector_vector_mul_sisd(int *x, int *y, int *z, int numel)
62  {
63
64      for (int i=0;i<numel;i++)
65      {
66          *(z+i)= (*(x+i))* (*(y+i));
67      }
68  }
69  //A SIMD vector multiplication
70  void vector_vector_mul_simd(int *x, int *y, int *z, int numel)
71  {
72      vint* vx = (vint*)x;
73      vint* vy = (vint*)y;
74      vint* vz = (vint*)z;
75
76      int reps=numel/VSIZE;
77      int remainder=numel%VSIZE;
78      for (int i=0;i<reps;i++)
79      {
80          vx=(vint*)(x+VSIZE*i);
81          vy=(vint*)(y+VSIZE*i);
82          vz=(vint*)(z+VSIZE*i);
83          (*vz)=(*vx) * (*vy);
84      }
85  }
86
87
88  //A SISD binary search
89  void binsearch(int*& p, int n, int key)
90  {
91      int step = 1 << (n-1);
92      p += step;
93      for (int i=0; i<(n-1); i++) chess_loop_range(1,) {
94          step = step >> 1;
95          if (key < *p)
96            p += step;
97          else
98            p -= step;
99      }
100     if (key < *p)
101       p += 1;
102 }
103
104 //A vector binary search using predication
```

```
105 void v_binsearch(chess_vector_ptr<vint>& p, int n, vint a)
106 {
107     int step = 1 << (n-1);
108     p += step;
109     for (int i=0; i<(n-1); i++) chess_loop_range(1,)
110     {
111         step >>= 1;
112         vint vstep;
113         vif (a < *p)
114             vstep = +step;
115         velse
116             vstep = -step;
117         vendif
118         p = p + vstep;
119 
120     }
121 
122     vif (a < *p)
123         p += 1;
124     vendif
125 }
126 //A SISD interpolation mechanism
127 void interpolate(int* X, long* Y, int n, int *xd, int* yd)
128 {
129     int* list = (int*)X;
130     int key=*xd;
131     int offset;
132     binsearch(list, n, key);
133 
134     offset=list-X;
135     (*yd) = (Y[offset-1]-Y[offset])/(X[offset]-X[offset-1])*((*xd)-X[
        offset])+Y[offset];
136 
137 }
138 //A SIMD interpolation mechanism
139 void v_interpolate(int* X, long* Y, int n, int *xd, int* yd)
140 {
141     int* list = (int*)X;
142     int key=*xd;
143     int offset;
144     binsearch(list, n, key);
145 
146     offset=list-X;
147 
148     (*yd) = (Y[offset-1]-Y[offset])/(X[offset]-X[offset-1])*((*xd)-X[
        offset])+Y[offset]; //LERP
149 
150 }
```

Appendix B

Paper

Efficient Hardware Mapping of LSTM Neural Networks for Automatic Speech Recognition

Georgios Evangelopoulos
ESAT/MICAS, KU Leuven, Belgium
School of Electrical & Computer Engineering, Aristotle University of Thessaloniki

***Abstract*—Recently Long Short-Term Memory Neural Networks (LSTMs) have become state-of-the-art models for a variety of machine learning tasks. However, LSTMs are usually very memory and computationally intensive. In order to enable LSTM-based classification in embedded systems such as smartphones and ubiquitous electronics for wireless sensor networks and the internet of things, their energy footprint should be reduced. This paper examines how LSTMs behave as different methods of approximate computing are applied. These algorithmic-level techniques are combined with architectural modifications to an Application Specific Instruction Set Processor (ASIP) to reduce energy consumption in an LSTM-based Automatic Speech Recognition system.**

I. Introduction

The different flavours of Neural Network architectures are generally considered as versatile tools for a plethora of machine learning tasks. In recent years, their popularity has increased due to the increased availability of computing power, since the computationally and memory intensive procedures of both training and evaluating neural architectures can be offloaded to multicore CPUs, GPUs or even computational clusters with small financial cost. With these advances, it has become possible for scientists to train and evaluate networks with an ever-increasing number of parameters.

Recurrent Neural Networks (RNNs) and specifically LSTMs are neural architectures that natively incorporate temporal context into both their training and evaluation, and have been recently used with big success in machine learning tasks that exhibit temporal dependencies.

Since LSTMs as the one in Figure 1 can natively incorporate temporal context, their network size can be kept relatively small compared to other similarly performing architectures that model context by various other methods. In parallel, researchers have been reporting that neural network architectures inherently exhibit error resilience. With these two concepts in mind, we perform an analysis that involves the design of an LSTM architecture, the application of approximate computing techniques and the derivation of a hardware architecture that reduces the energy consumption of LSTM applications.

Our contribution is twofold:

1) We show how methods such as a) precision scaling b) connection pruning and c) coarse approximations of the nonlinear activation functions influence the accuracy of a Bidirectional LSTM trained and evaluated on an Automatic Speech Recognition task using the TIMIT benchmark

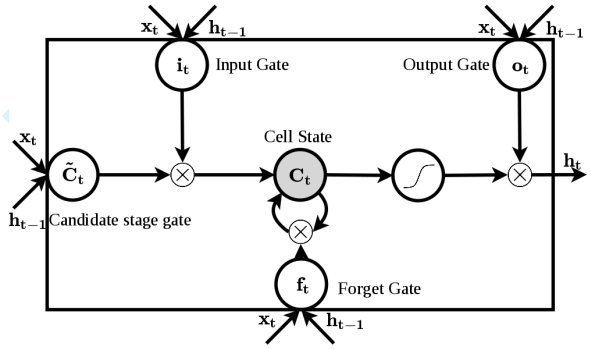


Fig. 1: An LSTM neural network. The current input $\mathbf{x_t}$ and the previous output $\mathbf{h_{t-1}}$ influence the internal state $\mathbf{C_t}$ which can be thought as the "memory". All three variables are combined to output a prediction for the current output, $\mathbf{h_t}$. The memory of the LSTM allows it to natively model very long temporal dependencies, making the architecture a prime candidate for sequential machine learning tasks.

2) We demonstrate the possible energy savings that can be achieved when using a customized hardware architecture that exploits the aforementioned methods.

This paper is structured as follows: In Section II we present the related work on LSTM, in Section III we present the LSTM architecture used for the Automatic Speech Recognition task, in Section IV we show how LSTMs can be improved to be less demanding with respect to their computational and memory requirements and how they can be made to be more suitable for embedded platforms. In Section V, we present modifications to an ASIP hardware architecture that can facilitate the evaluation of LSTM neural networks while reducing energy consumption. Finally, Section VI concludes this paper.

II. Related Work

Since the inception of the Long Short-Term Memory architecture, many LSTM variations have been devised and have become state-of-the-art solutions for a variety of sequential machine learning tasks, including Automatic Speech Recognition [1] [2], video and action classification [3] [4], unsegmented handwriting recognition and generation [5], polyphonic music modelling [6] and protein secondary structure prediction [7].

An LSTM has a smart structure consisting of four recurrent neural network layers. These are named the *input, forget,*

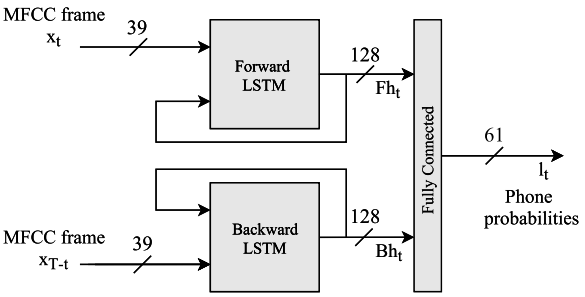


Fig. 2: High Level BLSTM architecture. Two LSTM networks work in parallel on the same input sequence, one processing it in a forward and one backward manner. This way, both previous and "future" context is taken into account for the prediction of each phone label in the sequence. The fully connected layer performs the final 1-of-N classification.

output and *candidate state gates*. The detail that sets them apart from other RNN architectures is the existence of an *internal state*, called the *cell state*. The outputs of the four gates depend on the current LSTM input and the previous LSTM output and regulate the amount of information that flows in and out of the cell state. The output of the LSTM block is dependent on both the current input and the internal state, as this has been formulated from all past inputs.

The ability of LSTMs to model arbitrarily long sequences with ease has stirred the interest of the machine learning community. Although various LSTM architectures have been developed with many record-breaking performances, a study about the error-resilience of LSTMs has not been yet completed. The effect however, of quantization error and connection pruning on the performance of fully connected neural networks has been documented in [8], [9], [10].

III. Neural Architecture Description

In this work, a Bidirectional LSTM architecture (BLSTM) for speech recognition similar to the one in [2] is trained for phone classification on the TIMIT dataset [11]. The high level outline of the architecture can be seen in Figure 2. In accordance with common practice in the literature [12], the inputs of the network consist of sequences of 39-dimensional vectors of MFCC coefficients, each calculated over 25ms of audio data with a shift of 10ms. The targets for the inputs are 1-of-61 phone labels according to the phonemic alphabet described in [13].

The network was modelled, trained and tested in the Torch framework [14]. The training method was Full Backpropagation Through Time [15]. The initial learning rate was set to $5 * 10^{-3}$ and it was incremented by 1% if the loss between two consecutive epochs declined and was decremented by 50% if the loss between two consecutive epochs rose. The performance metric was the Framewise Phone Classification Error (FCE) [2] over the TIMIT validation set and it complementary accuracy percentage ($Accuracy = (1 - FCE) * 100\%$).The size of the LSTM blocks was varied between 64, 128 and 256. The architecture achieved a peak accuracy of 69.8% when using an LSTM size of 128 while it achieved 70.2% accuracy when using LSTMs of size 256. The network with LSTM size 128 achieving 69.8% (Figure 2) is used as a baseline to monitor accuracy loss when exploring different approximate computing methods. We consider the training procedure to take place on a system with unlimited power budget and we only assess the accuracy loss of the evaluation, assuming no re-training.

The algorithm that corresponds to an LSTM forward pass can be shown in Figure 1. For the BLSTM architecture, it is assumed that the two LSTM layers are processing the same input sequence, one in a forward and one in a backward manner, while the output fully connected layer performs the final 1-of-61 classification.

Algorithm 1 LSTM Forward Pass

procedure FORWARDLSTM($\mathbf{X}$) ▷ Forward pass for a sequence $\mathbf{X}$ of input vectors $\mathbf{x_t}$
  $\mathbf{C_0} \leftarrow \mathbf{0}$
  $\mathbf{h_0} \leftarrow \mathbf{0}$
  for every sequence frame $\mathbf{x_t}$, $1 \leq t \leq T$ **do** ▷ T is the sequence size
    $\mathbf{f_t} = \sigma(\mathbf{W_f} \cdot \mathbf{x_t} + \mathbf{R_f} \cdot \mathbf{h_{t-1}} + \mathbf{b_f})$ ▷ Forget gate
    $\mathbf{i_t} = \sigma(\mathbf{W_i} \cdot \mathbf{x_t} + \mathbf{R_i} \cdot \mathbf{h_{t-1}} + \mathbf{b_i})$ ▷ Input gate
    $\mathbf{o_t} = \sigma(\mathbf{W_o} \cdot \mathbf{x_t} + \mathbf{R_o} \cdot \mathbf{h_{t-1}} + \mathbf{b_o})$▷ Output gate
    $\tilde{\mathbf{C_t}} = \tanh(\mathbf{W_C} \cdot \mathbf{x_t} + \mathbf{R_C} \cdot \mathbf{h_{t-1}} + \mathbf{b_C})$ ▷ Candidate state
    $\mathbf{C_t} = \mathbf{f_t} \odot \mathbf{C_{t-1}} + \mathbf{i_t} \odot \tilde{\mathbf{C_t}}$ ▷ The current state
    $\mathbf{h_t} = \mathbf{o_t} \odot \tanh(\mathbf{C_t})$ ▷ The current output
  end for
end procedure ▷ Output is stored in the sequence $\mathbf{H}$ of vectors $\mathbf{h_t}$

IV. Experiments

Structurally, the forward pass of an LSTM consists of matrix-vector multiplications, element-wise vector products, additions and application of squashing non-linear functions such as the logistic sigmoid $\sigma(x)$ and hyperbolic tangent $\tanh(x)$. From a parameter point of view, it consists of a set of "input" weight matrices $\mathbf{W}_\star$ where $\star$ denotes one of $\{i, o, f, C\}$ that are applied on the current input $\mathbf{x_t}$, a set of "recurrent" weight matrices $\mathbf{R}_\star$ which are applied on the previous LSTM output, $\mathbf{h_{t-1}}$, and a set of biases $\mathbf{b}_\star$. The Bidirectional LSTM of Figure 2 has parameter matrices $\mathbf{W}_\star$, $\mathbf{R}_\star$, $\mathbf{b}_\star$ for both the Forward and the Backward LSTM and also $\mathbf{W_{FC}}, \mathbf{b_{FC}}$ for the fully connected output layer.

For our experiments, the LSTM forward pass originating from the Torch Framework was reimplemented in MATLAB. By doing so, we obtained full control over each step of the algorithm and we were able to simulate quantization, connection pruning and variable look up table implementations of the nonlinear functions. All experiments are run on the validation set of the TIMIT benchmark. We report the relative accuracy obtained by each experiment, i.e. the ratio between

	$\times$	$+$
$\mathbf{f_t}$	$hd + h^2 + \sigma_1 h$	$hd + h^2 + 2h + \sigma_2 h$
$\mathbf{o_t}$	$hd + h^2 + \sigma_1 h$	$hd + h^2 + 2h + \sigma_2 h$
$\mathbf{i_t}$	$4hd + h^2 + \sigma_1 h$	$hd + h^2 + 2h + \sigma_2 h$
$\tilde{\mathbf{C}}_\mathbf{t}$	$hd + h^2 + t_1 h$	$hd + h^2 + 2h + t_2 h$
$\mathbf{C_t}$	$2h$	h
$\mathbf{h_t}$	$h + t_1 h$	$t_2 h$
Total	$4(hd + h^2)$ $+3\sigma_1 h + 2t_1 h + 3h$	$4(hd + h^2 + 2h)$ $+3\sigma_2 h + 2t_2 h + h$
$d = 39, h = 128$	$87,528$	$87,936$

TABLE II: Multiplications and additions in the LSTM forward pass. For a fixed input size d ($d = 39$ in the BLSTM architecture) the total number of operations in dominated by the network's size h ($h = 128$ in the BLSTM).

the accuracy obtained in an experiment and the architecture's baseline accuracy.

For each experiment we also present the energy savings that can be obtained in each case. The total power consumption of a digital circuit can be modelled as $P = P_{dyn} + P_{stat}$ where P_{dyn} represents the dynamic and P_{stat} the static power consumption. The dynamic power consumption can be modelled as $P_{dyn} = \alpha C f V_{dd}^2$ while P_{stat} is a static power consumption term influenced mainly by the leakage current of the transistors. The parameter that represents the switching activity, α, is influenced by the number of bits used in calculations used as seen Table I. The number of operations in a forward LSTM pass are presented in Table II. The energy modelling that is performed does not incorporate the energy spent in architecture-dependent operations and it can be thought of as the energy spent in all LSTM arithmetic (matrix-vector multiplications, element-wise vector multiplications and additions).

	$\times$	$+$	Reg.	Wire	SRAM
α	$\mathcal{O}(n^2)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(n_{max})$

TABLE I: Switching activity and elementary operations

It should be noted that while these techniques presented here can be used to reduce the energy consumption during evaluation, they impose no restrictions on training, which can be performed on any computing platform with any available set of resources and tools. This is because our primary concern is not the development, but rather the deployment of neural networks on embedded systems. A study that investigates the effect of limited numerical precision during the training of neural networks can be found in [16].

A. *Quantization*

On modern systems, LSTMs and other neural architectures usually run with 32 or even 64-bit floating point number representations. The computational requirements in terms of energy, time and hardware complexity of these operation in these are widely known [17] [18]. These representations

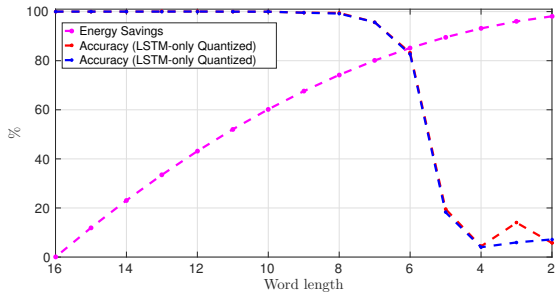


Fig. 3: The effect of quantization on LSTM accuracy and energy savings. Switching from 64 to 32-bit floating point has no effect on accuracy. Quantizing to 8-bit fixed point no relative accuracy loss is observed and further switching to 7-bit fixed-point retains a relative accuracy of 96%. A 7-bit LSTM network consumes 80% less power than its 16-bit counterpart.

might be optimal in terms of performance for general purpose CPUs or GPUs, but their wide range and high precision don't necessitate improvement on the *predictive performance* of the algorithms.

As mentioned in Table I, dramatic energy savings can be realized by reducing the number of bits used in calculations (e.g. in multipliers, precision scaling can gain $12\times$ in energy at 1%RMSE [19]). For that reason, in order to reduce energy consumption of the LSTM computations, we quantize the proposed architecture's weight matrices and the input data, using a fixed-precision word length WL. The goal is to monitor the accuracy loss that occurs due to this reduction of weight and input data resolution. In order to achieve best results, its important to consider the statistics of the network's parameters before performing quantization. Since large absolute values in the weight matrices indicate strong neural connections, it is wise to quantize in a way that avoids saturation. On the other hand, precision in parameters with absolute values around zero is considered less important. Under these assumptions, we rescale all parameters and input data with respect to the element with the largest absolute value, so that the limits of quantization interval match with the limits of the parameters or input data. This ensures a "best fit", allowing for both the necessary dynamic range and minimum achievable loss in precision.

According to the above method, we quantize $\mathbf{W}_\star$, $\mathbf{R}_\star$, $\mathbf{b}_\star$ for both LSTM blocks. We observe that switching from 64-bit floating point to 32-bit floating point is virtually harmless for the performance of the network, while when switching to fixed-precision, word lengths as low as $WL = 8$ can be used without a penalty in relative classification accuracy. Allowing for 96% relative accuracy, even 7 bits can be used. Based on the number of computations for the LSTM forward pass presented in Table II, we show that by switching from 16-bit fixed-point computations to 8 or 7 bits, we can achieve up to 74% and 80% energy savings. The results are shown in Figure 3.

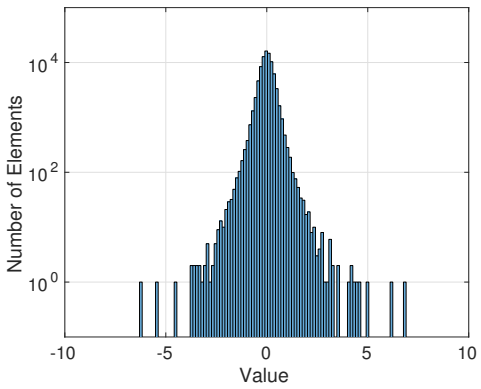


Fig. 4: Forward LSTM parameter distribution. The vast majority of parameters are very close to zero, but there exist exceptions that are even 17σ away from the mean. The Backward LSTM and the fully connected layers exhibit similar distributions. Note the log-scale on the y-axis.

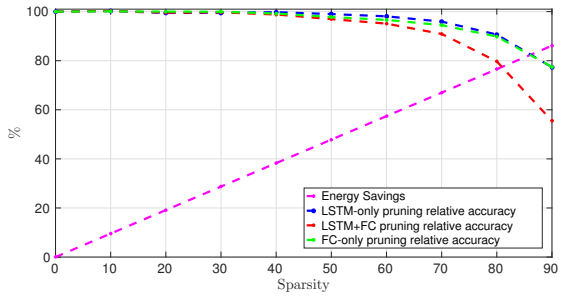


Fig. 5: Connection pruning and energy gain. LSTM-only pruning allows for up to 40% zeroed parameters with no accuracy loss, and then performance decreases slowly, allowing up to 70% pruning with 96.1% accuracy compared to the baseline. FC-only pruning shows similar behaviour, but pruning of both layers leads to 95% relative accuracy at 60% pruning and then leads to significant deterioration. Energy saving due to pruning are shown in magenta.

B. Connection Pruning

As observed in [20], neural networks tend to be vastly overparameterized architectures, and various methods have been developed to reduce the number of network weights. Since modern neural network architectures tend to have a number of parameters that can easily extend to the millions, it is apparent that the evaluation of these networks is a very memory intensive task. Our strategy here is to investigate how connection pruning affects LSTM performance and also to model what impact connection pruning will have in energy consumption.

Figure 4 shows the distribution of the Forward LSTM parameters. The distribution is relatively narrow and zero-centered but has a large amount of outlier values, even up to 17σ away from the mean. Intuitively, one can understand that the larger the absolute value of a weight, the stronger the connection between two neurons is, and accordingly, the more "important" the connection for the process of inference. From a statistical point of view, this points to the fact that neural network performance and behaviour is governed by the outliers in the network parameter distribution.

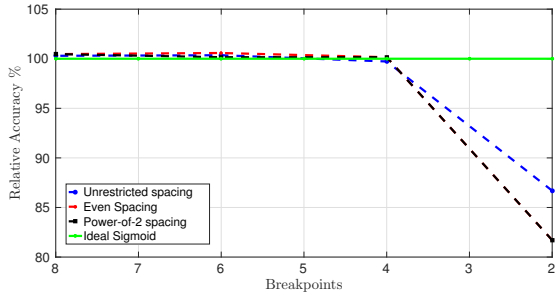
To model connection pruning, all connections with an absolute value under a specific "importance" threshold regarded as redundant and their weights are set to zero. This operation equals to artificially increasing the sparsity of the weight matrices, and conceptually corresponds to decreasing the coupling among the system's basic functional units (neurons).

Since each matrix has its own statistical properties, it would not be "fair" to prune all connections under one specific threshold value. Instead, in this experiment, the sparsity of the matrices is considered the independent variable and a per-matrix threshold is calculated. Based on this thresholding, all the elements of each matrix having a smaller absolute value than the matrix-specific threshold are zeroed. With this rationale, it is easier to quantify the effect of connection pruning over LSTM accuracy. We show that with 40% of connections pruned there is no loss in prediction accuracy, while pruning to 70% reduces relative accuracy to 96%. It is worth noting that connection pruning on the fully connected layer is more "damaging" to the network, probably because it recognizes higher-level features than the LSTM blocks. Luckily, the number of its parameters is rather small compared to the overall design (only 8% of the overall parameters belong to the 256×61 layer) and therefore, there is incentive to *avoid* pruning of the output layer or enforce *lighter* pruning.

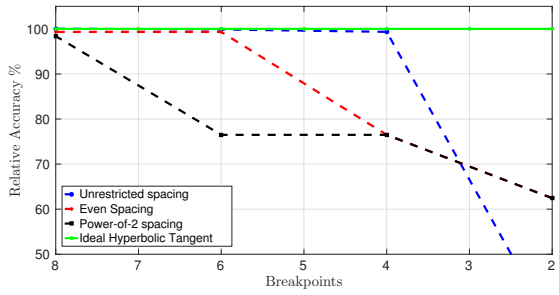
From the energy consumption standpoint, we investigate how the *skipping* of unnecessary computations impacts energy savings. Computation skipping is complicated to achieve in software, since the extra control needed for the checking would nullify the overall attempt at energy reduction. However, since the networks parameters are known and fixed, unimportant connections can be set to zero and hardware checking can eliminate the switching of multipliers and adders when the result of the operation is known beforehand. Based on Table II we identify what percentage of LSTM arithmetic involves pre-known parameters and can therefore be skipped efficiently. For the network under consideration (with input size $d = 39$, LSTM size $h = 128$ and output size $l = 61$) more than 97% of the operations arise from matrix-vector products with known parameters. On these grounds, we show that a 40% and a 70% pruned network use 38% and 67% less energy than an unpruned. The results are shown in Figure 5.

C. LUT-based implementations of non-linearities

In Table II the number of multiplications and additions needed for the application of non-linearities are modelled by the factors σ_1, t_1 and σ_2, t_2 respectively. In this work, we implement look-up table versions while varying the breakpoint number and spacing. The best energy/precision tradeoff occurs when a *power-of-2* implementation is used. Evaluating the nonlinear functions at points that are powers of two allows for



(a) $\sigma(x)$ activation function



(b) $\tanh(x)$ activation function

Fig. 6: The effect of coarse activation functions on accuracy. The three cases evaluated are *unrestricted breakpoint spacing* (blue), *even spacing* (red), *power-of-2 spacing* (black). In (a) it is shown that replacing the sigmoid function even with the cheapest, easiest to implement *power-of-2* approximation with 4 breakpoints, no loss in accuracy is observed. In (b), approximating $\tanh(x)$ with the same power-of-2 strategy requires 8 breakpoints to achieve perfect relative accuracy.

a very simple implementation of the interpolation mechanism using one shift, one bit-masking operation, one multiplication ($\sigma_1 = t_1 = 1$) and two additions ($\sigma_2 = t_2 = 2$). Other approaches that require fewer breakpoints but require more complex interpolation routines were also benchmarked. Figure 6 shows that even with the simplest of approximations (power-of-two) no relative accuracy loss is introduced when using 8 breakpoints. We consider this to be of great importance, since it shows that the complicated LSTM structure can function correctly even when its activation functions are far from ideal.

V. Implementation

In section IV we showed that LSTMs do not suffer from significant performance penalty, even when their computational components and parameters are replaced with aggressive approximations. Here, we present a hardware architecture that exploits the aforementioned quality, using a custom word length ASIP-based approach.

The reduced word length of the data path realizes the energy savings mentioned in section IV-A. A vector processing unit is used to exploit the inherent data-level parallelism available in neural networks, accelerating the matrix-vector

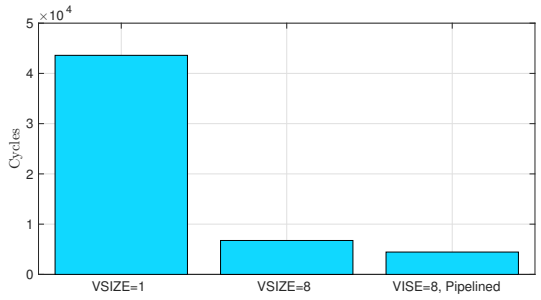


Fig. 7: The cycle savings that are achieved during a Matrix-Vector multiplication of size $(48 \times 128) \times (128 \times 1)$ using the SIMD and pipelining combination over a Single Instruction Single Data (SISD) architecture.

multiplications, performing parallel element-wise vector multiplications and additions and parallel evaluation of the non-linear activation functions.

Furthermore, we augment the ASIP instruction set with instructions that allow for the load of two vectors from the memory, their element-wise multiplication and the summation of the partial products in one clock cycle. To achieve this, we modify the memory accessing and we define new pipeline stages and transitories. The modified ASIP architecture is shown in Figure 8.

The usage of SIMD instructions through the vector processing unit and the definition of extra pipeline stages reduce the overall amount of cycles necessary for the LSTM forward pass. This allows for Dynamic Frequency and Voltage Scaling (DVFS) and can be exploited for more energy savings. In order to exploit this architecture efficiently and to perform profiling, the basic computational kernels for the LSTM forward pass are rewritten in C. The gain in clock cycles during matrix-vector multiplications (which according to the analysis of section IV-B, account for 97% of all operations in the forward pass) are shown in Figure 7. Besides the gains realized in actual *cycles*, which are simulated with cycle-accurate simulators built for the custom ASIP, the RTL hardware description of the vector multiplier is modified to perform hardware zero-checking. While this cannot be modelled by the instruction set simulator, it leads to significant savings (presented in section IV-B) when connection pruning takes place, since it reduces the switching activity of the custom vector multiplier.

VI. Conclusion

In this work, we developed a Long Short-Term Memory based neural architecture that was benchmarked over the sequential machine learning task of Automatic Speech Recognition, using the TIMIT dataset. After obtaining our baseline results, we explored the loss in accuracy that occurs when approximate computing methods are applied on the architecture's inference process. These methods include quantizing to short word lengths, pruning LSTM connections and coarsely approximating non-linear activation functions. The motivation behind this exploration is that approximate computing can

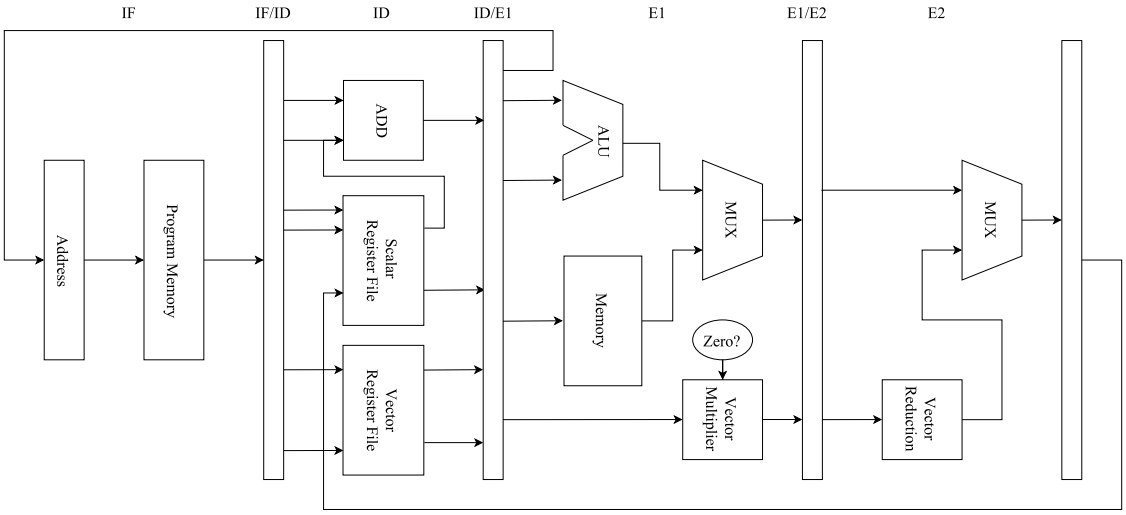


Fig. 8: The modified ASIP architecture. This architecture has the following modifications: a) It incorporates a vector register file and a vector multiplier that has hardware checking to avoid switching if any operand is zero and b) It has an extra pipeline stage E2 where the vector reduction is executed before it is written back in the register file. With this structure, a vector-vector inner product can be achieved in exactly one clock cycle, allowing for acceleration of the matrix-vector product process.

lead to significant energy savings, which will in turn facilitate the deployment of machine learning solutions on embedded systems with a limited energy budget.

We show that LSTMs suffer no accuracy loss when using down to 8 bits of precision for their computations instead of the more general 64 or 32 bits, that pruning up to 70% percent of their connections results in less than 4% relative accuracy loss and that efficient LUT-approximations of the activation functions with as few as 8 precomputed values give no performance penalty to the inference process. We estimate 80% energy reduction (with quantization to 7-bits) and 67% energy reduction (with pruning to 70%) over the energy baseline. Finally, we propose a hardware architecture that can exploit these approximate computing techniques to achieve the possible energy savings.

REFERENCES

[1] A. Graves and N. Jaitly, "Towards End-To-End Speech Recognition with Recurrent Neural Networks," *JMLR Workshop and Conference Proceedings*, vol. 32, no. 1, pp. 1764–1772, 2014. [Online]. Available: http://jmlr.org/proceedings/papers/v32/graves14.pdf

[2] 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.

[3] J. Donahue, L. Anne Hendricks, S. Guadarrama, M. Rohrbach, S. Venugopalan, K. Saenko, and T. Darrell, "Long-term recurrent convolutional networks for visual recognition and description," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2015, pp. 2625–2634.

[4] N. Srivastava, E. Mansimov, and R. Salakhutdinov, "Unsupervised learning of video representations using lstms," *arXiv preprint arXiv:1502.04681*, 2015.

[5] A. Graves, "Generating sequences with recurrent neural networks," *arXiv preprint arXiv:1308.0850*, pp. 1–43, 2013. [Online]. Available: http://arxiv.org/abs/1308.0850

[6] N. Boulanger-Lewandowski, Y. Bengio, and P. Vincent, "Modeling temporal dependencies in high-dimensional sequences: Application to polyphonic music generation and transcription," *arXiv preprint arXiv:1206.6392*, 2012.

[7] S. K. Sønderby and O. Winther, "Protein secondary structure prediction with long short term memory networks," *arXiv preprint arXiv:1412.7828*, 2014.

[8] M. Jiang and G. Gielen, "The effects of quantization on multi-layer feedforward neural networks," *International journal of pattern recognition and artificial intelligence*, vol. 17, no. 04, pp. 637–661, 2003.

[9] G. Dundar and K. Rose, "The effects of quantization on multilayer neural networks." *IEEE transactions on neural networks/a publication of the IEEE Neural Networks Council*, vol. 6, no. 6, pp. 1446–1451, 1994.

[10] Z. Du, A. Lingamneni, Y. Chen, K. Palem, O. Temam, and C. Wu, "Leveraging the Error Resilience of Neural Networks for Designing Highly Energy Efficient Accelerators," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 34, no. 8, pp. 1223–1235, 2015.

[11] J. S. Garofolo, L. F. Lamel, W. M. Fisher, J. G. Fiscus, and D. S. Pallett, "DARPA TIMIT acoustic-phonetic continous speech corpus CD-ROM. NIST speech disc 1-1.1," *NASA STI/Recon Technical Report N*, vol. 93, Feb. 1993.

[12] C. Lopes and F. Perdigão, "Phone recognition on the TIMIT database," *Speech Technologies/Book*, vol. 1, pp. 285–302, 2011.

[13] S. Seneff and V. Zue, "Transcription and alignment of the timit database," *TIMIT CD-ROM Documentation*, 1988.

[14] R. Collobert, S. Bengio, and J. Mariéthoz, "Torch: a modular machine learning software library," IDIAP, Tech. Rep., 2002.

[15] P. J. Werbos, "Backpropagation through time: what it does and how to do it," *Proceedings of the IEEE*, vol. 78, no. 10, pp. 1550–1560, 1990.

[16] S. Gupta, A. Agrawal, K. Gopalakrishnan, and P. Narayanan, "Deep Learning with Limited Numerical Precision," *arXiv:1502.02551 [cs, stat]*, vol. 37, 2015. [Online]. Available: http://arxiv.org/abs/1502.02551$\delimiter"026E30F$nhttp://www.arxiv.org/pdf/1502.02551.pdf

[17] J. L. Hennessy and D. A. Patterson, *Computer architecture: a quantitative approach*. Elsevier, 2011.

[18] G. Govindu, L. Zhuo, S. Choi, and V. Prasanna, "Analysis of high-performance floating-point arithmetic on fpgas," in *Parallel and Distributed Processing Symposium, 2004. Proceedings. 18th International*. IEEE, 2004, p. 149.

[19] B. Moons, B. De Brabandere, L. Van Gool, and M. Verhelst, "Energy-efficient convnets through approximate computing," *arXiv preprint arXiv:1603.06777*, 2016.

[20] J. Pool and W. J. Dally, "Neural Networks," pp. 1–9.

Αποτελεσματικές Υλοποιήσεις Νευρωνικών Δικτύων LSTM για Αυτόματη Αναγνώριση Ομιλίας

Georgios Evangelopoulos
ESAT/MICAS, KU Leuven, Belgium
School of Electrical & Computer Engineering, Aristotle University of Thessaloniki

***Περίληψη*—Τελευταία, τα Long Short-Term Memory Νευρωνικά Δίκτυα (LSTM) επιδεικνύουν πρωτοφανείς επιδόσεις σε μια πληθώρα προβλημάτων μάθησης των μηχανών. Ωστόσο, τα LSTM είναι, συνήθως, ιδιαίτερα απαιτητικά σε υπολογιστική ισχύ αλλά και μνήμη. Με σκοπό να καταστεί δυνατή η χρήση LSTM σε ενσωματωμένα συστήματα όπως smartphones και "πανταχού παρόντα" ηλεκτρονικά σε ασύρματα δίκτυα αισθητήρων και στο επερχόμενο Διαδίκτυο των Πραγμάτων (Internet of Things, IoT), η ενεργειακή τους κατανάλωση θα πρέπει να μειωθεί σημαντικά. Η δημοσίευση αυτή εξετάζει την συμπεριφορά των LSTM κατά την εφαρμογή σε αυτά διαφόρων "προσεγγιστικών" υπολογιστικών μεθόδων (approximate computing). Αυτές οι αλγοριθμικές τεχνικές συνδυάζονται με τροποποιήσεις της αρχιτεκτονικής ενός επεξεργαστή ειδικής χρήσης (ASIP) με στόχο την μείωση της ενεργειακής κατανάλωσης ενός συστήματος Αυτόματης Αναγνώρισης Ομιλίας βασισμένο σε νευρωνικά δίκτυα LSTM.**

I. Εισαγωγή

Οι διάφορες παραλλαγές των αρχιτεκτονικών Νευρωνικών Δικτύων θεωρούνται εν γένει ευέλικτα εργαλεία για την επίλυση μια πληθώρας προβλημάτων νοημοσύνης των μηχανών. Τα τελευταία χρόνια, η δημοτικότητά τους έχει αυξηθεί εξαιτίας της αυξημένης διαθεσιμότητας υπολογιστικών πόρων. Πλέον, οι διαδικασίες της εκπαίδευσης και της χρήσης νευρωνικών δικτύων μπορεί να εκτελεστεί σε πολυπύρηνους επεξεργαστές, κάρτες γραφικών και σε διανεμημένα υπολογιστικά συστήματα με μικρό οικονομικό κόστος. Με τις εξελίξεις αυτές, έχει καταστεί δυνατή η υλοποίηση και χρήση νευρωνικών δικτύων με ολοένα και περισσότερες παραμέτρους.

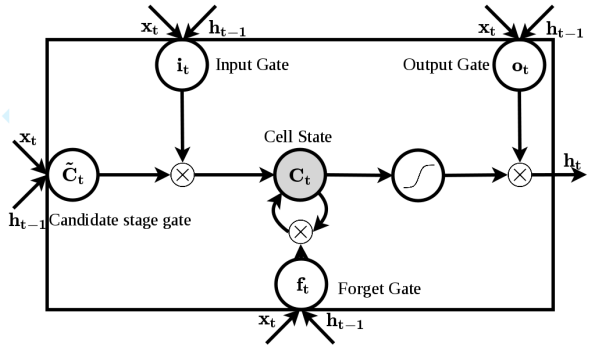
Τα νευρωνικά δίκτυα με αναδράσεις (Recurrent Neural Networks, RNN) και συγκεκριμένα τα LSTM είναι νευρωνικές αρχιτεκτονικές που έχουν την εγγενή δυνατότητα να ενσωματώνουν πληροφορίες από την χρονική συνιστώσα (το χρονικό πλαίσιο) τόσο κατά την εκπαίδευση τους όσο και κατά την κανονική λειτουργία τους. Η ιδιότητά τους αυτή έχει οδηγήσει στην επιτυχή χρήση τους σε προβλήματα μάθησης μηχανών των οποίων τα δεδομένα παρουσιάζουν χρονικές συσχετίσεις.

Καθώς τα LSTM όπως αυτό του σχήματος 1 μπορούν να ενσωματώνουν εγγενώς πληροφορία από τον άξονα του χρόνου, το μέγεθός τους (από άποψη αριθμού παραμέτρων) μπορεί να κρατηθεί πολύ μικρότερο σε σχέση με άλλες νευρωνικές αρχιτεκτονικές που χρησιμοποιούν άλλους τρόπους για να μοντελοποιήσουν χρονικές εξαρτήσεις. Παράλληλα, η έρευνα στα νευρωνικά δίκτυα έχει δείξει ότι οι νευρωνικές αρχιτεκτονικές επιδεικνύουν μια έμφυτη ανθεκτικότητα όσον αφορά τα λάθη στους αριθμητικούς τους υπολογισμούς. Βασιζόμενοι σε αυτές τις δύο παρατηρήσεις, πραγματοποιούμε μια ανάλυση που αποτελείται από τον σχεδιασμό και την υλοποίηση μιας νευρωνικής LSTM αρχιτεκτονικής, από την εφαρμογή σε αυτήν τεχνικών *κατά-προσέγγιση επεξεργασίας δεδομένων* και από την σχεδίαση μιας αρχιτεκτονικής υλικού η οποία ελαχιστοποιεί την ενεργειακή κατανάλωση των αλγορίθμων που αξιοποιούν LSTM νευρωνικά δίκτυα.

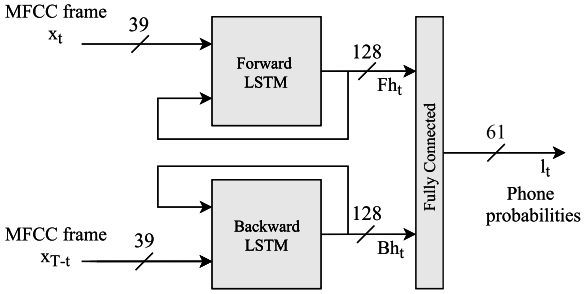
Η συνεισφορά μας είναι διπλή:

1) Δείχνουμε πώς μέθοδοι όπως α) η διαβάθμιση της χρησιμοποιούμενης υπολογιστικής ακρίβειας β) το "κλάδεμα" των νευρωνικών συνάψεων και γ) η χρήση προσεγγίσεων των μη γραμμικών συναρτήσεων ενεργοποίηση επηρεάζουν την ακρίβεια ενός αμφίδρομου LSTM (Bidirectional LSTM) το οποίο έχει εκπαιδευτεί πάνω στο πρόβλημα της Αυτόματης Αναγνώρισης Ομιλίας, χρησιμοποιώντας το σύνολο δεδομένων TIMIT.
2) Επιδεικνύουμε την ελαχιστοποίηση της ενεργειακής κατανάλωσης που μπορεί να επιτευχθεί με την χρήση ενός επεξεργαστή ειδικής χρήσης που έχει βελτιστοποιηθεί ώστε να εκμεταλλεύεται τις παραπάνω μεθόδους.

Η δημοσίευση αυτή είναι δομημένη ως εξής: Στην ενότητα II παρουσιάζουμε τη σχετική έρευνα πάνω στα LSTM, στην ενότητα III παρουσιάζουμε την νευρωνική LSTM αρχιτεκτονική που χρησιμοποιούμε για την επίλυση του προβλήματος της Αυτόματης Αναγνώρισης Ομιλίας, στην ενότητα IV δείχνουμε πως τα νευρωνικά δίκτυα LSTM μπορούν να βελτιωθούν ως προς τις απαιτήσεις τους σε υπολογιστική ισχύ και μνήμη και πως μπορούν να τροποποιηθούν ώστε να γίνουν πιο κατάλληλα για τη χρήση τους σε ενσωματωμένα συστήματα. Στην ενότητα V, παρουσιάζουμε τις τροποποιήσεις σε μία αρχιτεκτονική ενός επεξεργαστή ειδικής χρήσης (ASIP) που μπορούν να διευκολύνουν το ευθύ πέρασμα του νευρωνικού δικτύου ελαχιστοποιώντας την ενεργειακή του κατανάλωση. Η ενότητα VI αποτελεί την σύνοψη αυτής της



Σχήμα 1: Ένα νευρωνικό δίκτυο LSTM. Η τρέχουσα είσοδος $\mathbf{x_t}$ και η προηγούμενη έξοδος $\mathbf{h_{t-1}}$ επηρεάζουν την εσωτερική κατάσταση $\mathbf{C_t}$ η οποία μπορεί να θεωρηθεί ως κάποιου είδους μνήμη. Και οι τρεις μεταβλητές συνδυάζονται για την παραγωγή μιας πρόβλεψης για την τρέχουσα είσοδο, $\mathbf{h_t}$. Η μνήμη των LSTM τους επιτρέπει να μοντελοποιούν εγγενώς χρονικές εξαρτήσεις που απέχουν πολύ μεταξύ τους, καθιστώντας την αρχιτεκτονική αυτή πρωταρχικό υποψήφιο για πολλά ακολουθιακά προβλήματα μάθησης μηχανών.



Σχήμα 2: Αρχιτεκτονική BLSTM υψηλού επιπέδου. Δύο δίκτυα LSTM επεξεργάζονται παράλληλα την ίδια ακολουθία εισόδου, με το ένα να την επεξεργάζεται προς τα εμπρός και το άλλο προς τα πίσω. Με τον τρόπο αυτό, τόσο τα παρελθοντικά όσο και τα μελλοντικά "συμφραζόμενα" λαμβάνονται υπόψιν για την πρόβλεψη κάθε φωνήματος στην ακολουθία. Το πλήρως διασυνδεδεμένο επίπεδο πραγματοποιεί την τελική 1-από-61 ταξινόμηση.

δημοσίευσης.

II. Σχετικές έρευνες

Από την αρχική επινόηση των αρχιτεκτονικών LSTM, πολλές παραλλαγές έχουν αναπτυχθεί και έχουν αποτελέσει τις καλύτερες λύσεις σε μια μεγάλη ποικιλία ακολουθιακών προβλημάτων νοημοσύνης των μηχανών, συμπεριλαμβανομένης της Αυτόματης Αναγνώρισης Ομιλίας [1] [2], της ταξινόμησης βίντεο και ενεργειών [3] [4], της αναγνώρισης και δημιουργίας μη κατατετμημένου γραφικού χαρακτήρα [5], της μοντελοποίησης πολυφωνικής μουσικής και της πρόβλεψης πρωτεϊνικών δομών [6].

Ένα νευρωνικό δίκτυο LSTM αποτελείται από τέσσερα επίπεδα νευρωνικών δικτύων με αναδράσεις. Αυτά τα επίπεδα ονομάζονται πύλες *εισόδου, εξόδου, διαγραφής* και *υποψήφιας κατάστασης*. Η λεπτομέρεια που καθιστά τα LSTM διαφορετικά από άλλες αρχιτεκτονικές με αναδράσεις είναι η ύπαρξη μιας *εσωτερικής κατάστασης*. Οι έξοδοι των τεσσάρων πυλών εξαρτώνται τόσο από τα τρέχοντα δεδομένα στην είσοδο του δικτύου όσο και από τις προηγούμενες εξόδους του δικτύου. Οι έξοδοι αυτές ελέγχουν το ποσό της πληροφορίας που ρέει προς και από την εσωτερική κατάσταση του δικτύου. Η συνολική έξοδος του δικτύου εξαρτάται τόσο από τα τρέχοντα δεδομένα στην είσοδο όσο και από την εσωτερική κατάσταση, όπως αυτή έχει εξελιχθεί από όλα τα προηγούμενα δεδομένα εισόδου.

Η ικανότητα των LSTM να μοντελοποιούν εύκολα ακολουθίες δεδομένων αυθαιρέτου μήκους έχει εξάψει το ενδιαφέρον της επιστημονικής κοινότητας της νοημοσύνης των μηχανών. Αν και έχουν υλοποιηθεί πολλές αρχιτεκτονικές LSTM, συχνά με κορυφαίες επιδόσεις, μια μελέτη για την ανθεκτικότητά τους σε αριθμητικά σφάλματα δεν έχει ακόμα πραγματοποιηθεί. Οι συνέπειες, ωστόσο, του κλαδέματος νευρωνικών συνάψεων για τα νευρωνικά δίκτυα χωρίς αναδράσεις έχουν, ως ένα βαθμό, μελετηθεί [7], [8], [9].

III. Περιγραφή της Νευρωνικής Αρχιτεκτονικής

Σε αυτή την έρευνα, μια αμφίδρομη αρχιτεκτονική LSTM (BLSTM) για αναγνώριση ομιλίας παρόμοια με αυτήν που παρουσιάζεται στο [2] εκπαιδεύεται για ταξινόμηση φωνημάτων στο σύνολο δεδομένων TIMIT [10]. Η αρχιτεκτονική αυτή παρουσιάζεται στο Σχήμα 2. Ακολουθώντας την κοινή πρακτική της βιβλιογραφίας [11], οι είσοδοι του νευρωνικού δικτύου αποτελούνται από ακολουθίες 39-διάστατων διανυσμάτων από συντελεστές MFCC, υπολογισμένων πάνω σε ηχητικά δεδομένα, με ένα χρονικό παράθυρο 25ms και με ολίσθηση 10ms. Οι κλάσεις της ταξινόμησης είναι τα 61 φωνήματα σύμφωνα με το φωνημικό αλφάβητο που παρουσιάζεται στο [12].

Το νευρωνικό δίκτυο σχεδιάστηκε, εκπαιδεύτηκε και αξιολογήθηκε στο περιβάλλον Torch [13]. Η μέθοδος εκπαίδευσης ήταν η πλήρης οπισθοδιάδοση μέσα στο χρόνο (Full Backpropagation Through Time) [14]. Ο αρχικός ρυθμός μάθησης (learning rate) ορίστηκε στο $5 * 10^{-3}$ και αυξανόταν κατά 1% αν η απώλεια μεταξύ δύο διαδοχικών εποχών μειωνόταν, ενώ ελαττωνόταν κατά 50% αν η απώλεια μεταξύ δύο διαδοχικών εποχών αυξανόταν. Η μετρική της επίδοσης ήταν το σφάλμα ταξινόμησης φωνημάτων ανά πλαίσιο (Framewise Phone Classification Error, FCE) [2] στο σύνολο επαλήθευσης(validation set) και το συμπληρωματικό του μέγεθος, η ακρίβεια ($Accuracy = (1 - FCE) * 100\%$). Το μέγεθος των δικτύων LSTM κυμάνθηκε μεταξύ 64, 128 και 256. Η αρχιτεκτονική πέτυχε μέγιστη ακρίβεια 69.8% όταν χρησιμοποιήθηκε ένα δίκτυο μεγέθους 128 και ακρίβεια 70.2% όταν χρησιμοποιήθηκε δίκτυο μεγέθους 256. Η αρχιτεκτονική με το δίκτυο μεγέθους 128 που επιτυγχάνει ακρίβεια 69.8% (Σχήμα 2) χρησιμοποιείται ως σημείο αναφοράς για τη μελέτη της απώλειας ακρίβειας κατά την διερεύνηση διάφορων προσεγγιστικών μεθόδων επεξεργασίας

δεδομένων (approximate computing). Θεωρούμε πως η διαδικασία της εκπαίδευσης λαμβάνει χώρα σε ένα σύστημα με απεριόριστη διαθέσιμη ισχύ και αξιολογούμε μόνο την απώλεια ακρίβειας κατά το ευθύ πέρασμα, θεωρώντας πως το δίκτυο δεν επανεκπαιδεύεται.

Ο αλγόριθμος που αντιστοιχεί στο ευθύ πέρασμα φαίνεται στο Σχήμα 1. Για την αμφίδρομη αρχιτεκτονική υποτίθεται πως τα δύο LSTM επεξεργάζονται την ίδια ακολουθία εισόδου, το ένα προς τα εμπρός ενώ το άλλο προς τα πίσω, ενώ το επίπεδο εξόδου, αποτελούμενο από ένα πλήρως διασυνδεδεμένο νευρωνικό δίκτυο εκτελεί την τελική 1-από-61 ταξινόμηση.

Algorithm 1 LSTM Forward Pass

procedure ForwardLSTM($\mathbf{X}$) ▷ Forward pass for a sequence $\mathbf{X}$ of input vectors $\mathbf{x_t}$
  $\mathbf{C_0} \leftarrow \mathbf{0}$
  $\mathbf{h_0} \leftarrow \mathbf{0}$
  for every sequence frame $\mathbf{x_t}$, $1 \leq t \leq T$ **do** ▷ T is the sequence size
    $\mathbf{f_t} = \sigma(\mathbf{W_f} \cdot \mathbf{x_t} + \mathbf{R_f} \cdot \mathbf{h_{t-1}} + \mathbf{b_f})$ ▷ Forget gate
    $\mathbf{i_t} = \sigma(\mathbf{W_i} \cdot \mathbf{x_t} + \mathbf{R_i} \cdot \mathbf{h_{t-1}} + \mathbf{b_i})$ ▷ Input gate
    $\mathbf{o_t} = \sigma(\mathbf{W_o} \cdot \mathbf{x_t} + \mathbf{R_o} \cdot \mathbf{h_{t-1}} + \mathbf{b_o})$ ▷ Output gate
    $\mathbf{\tilde{C}_t} = \tanh(\mathbf{W_C} \cdot \mathbf{x_t} + \mathbf{R_C} \cdot \mathbf{h_{t-1}} + \mathbf{b_C})$ ▷ Candidate state
    $\mathbf{C_t} = \mathbf{f_t} \odot \mathbf{C_{t-1}} + \mathbf{i_t} \odot \mathbf{\tilde{C}_t}$ ▷ The current state
    $\mathbf{h_t} = \mathbf{o_t} \odot \tanh(\mathbf{C_t})$ ▷ The current output
  end for
end procedure ▷ Output is stored in the sequence $\mathbf{H}$ of vectors $\mathbf{h_t}$

IV. Πειραματικό Μέρος

Δομικά, το ευθύ πέρασμα ενός LSTM αποτελείται από πολλαπλασιασμούς πινάκων με διανύσματα, πολλαπλασιασμούς διανυσμάτων στοιχείο προς στοιχείο, διανυσματικές προσθέσεις και εφαρμογές μη γραμμικών συναρτήσεων ενεργοποίησης όπως η σιγμοειδής $\sigma(x)$ και η υπερβολική εφαπτομένη $\tanh(x)$. Από άποψη παραμέτρων, αποτελείται από ένα σύνολο από πίνακες με βάρη εισόδου $\mathbf{W}_\star$ όπου ο συμβολισμός $\star$ δηλώνει ένα από τα $\{i, o, f, C\}$ που εφαρμόζονται στο διάνυσμα εισόδου $\mathbf{x_t}$, ένα σύνολο από πίνακες με βάρη ανάδρασης $\mathbf{R}_\star$ που εφαρμόζονται στην προηγούμενη έξοδο του νευρωνικού δικτύου $\mathbf{h_{t-1}}$ και ένα σύνολο από διανύσματα προδιάθεσης $\mathbf{b}_\star$. Το αμφίδρομο LSTM του σχήματος 2 έχει παραμέτρους $\mathbf{W}_\star$, $\mathbf{R}_\star$, $\mathbf{b}_\star$ τόσο για το προς-τα-μπροστά LSTM όσο και για το προς-τα-πίσω LSTM και παραμέτρους $\mathbf{W_{FC}}, \mathbf{b_{FC}}$ για το πλήρως διασυνδεδεμένο επίπεδο εξόδου.

Για τα πειράματα που πραγματοποιήσαμε, το ευθύ πέρασμα του LSTM, αρχικά υλοποιημένο στο περιβάλλον Torch, επαναϋλοποιήθηκε στο MATLAB. Με τον τρόπο αυτό, αποκτήσαμε πλήρη έλεγχο σε κάθε βήμα του αλγορίθμου και είχαμε τη δυνατότητα να προσομοιώσουμε κβαντισμό παραμέτρων, κλάδεμα συνάψεων καθώς και διαφορετικές υλοποιήσεις των συναρτήσεων ενεργοποίησης. Όλα τα πειράματα

	$\times$	$+$
$\mathbf{f_t}$	$hd + h^2 + \sigma_1 h$	$hd + h^2 + 2h + \sigma_2 h$
$\mathbf{o_t}$	$hd + h^2 + \sigma_1 h$	$hd + h^2 + 2h + \sigma_2 h$
$\mathbf{i_t}$	$4hd + h^2 + \sigma_1 h$	$hd + h^2 + 2h + \sigma_2 h$
$\mathbf{\tilde{C}_t}$	$hd + h^2 + t_1 h$	$hd + h^2 + 2h + t_2 h$
$\mathbf{C_t}$	$2h$	h
$\mathbf{h_t}$	$h + t_1 h$	$t_2 h$
Total	$4(hd + h^2)$ $+3\sigma_1 h + 2t_1 h + 3h$	$4(hd + h^2 + 2h)$ $+3\sigma_2 h + 2t_2 h + h$
$d = 39, h = 128$	$87,528$	$87,936$

Πίνακας II: Πολλαπλασιασμοί και προσθέσεις κατά το ευθύ πέρασμα του LSTM. Για μια συγκεκριμένη διάσταση εισόδου d ($d = 39$ στην αρχιτεκτονική BLSTM) ο συνολικός αριθμός των υπολογισμών εξαρτάται κυρίως από το μέγεθος του δικτύου h ($h = 128$ στην αρχιτεκτονική BLSTM).

χρησιμοποιούν το σύνολο επαλήθευσης του TIMIT. Αναφέρουμε την σχετική ακρίβεια που επιτυγχάνεται με κάθε πείραμα, δηλ. το λόγο της ακρίβειας που επετεύχθη στο πείραμα ως προς την ακρίβεια της αρχιτεκτονικής αναφοράς.

Για κάθε πείραμα, παρουσιάζουμε επίσης την επιτεύξιμη εξοικονόμηση ενέργειας κάθε περίπτωσης. Η συνολική κατανάλωση ισχύος ενός ψηφιακού κυκλώματος μπορεί να μοντελοποιηθεί ως $P = P_{dyn} + P_{stat}$ όπου P_{dyn} η δυναμική και P_{stat} η στατική κατανάλωση ισχύος. Η δυναμική κατανάλωση ισχύος μπορεί να μοντελοποιηθεί ως $P_{dyn} = \alpha C f V_{dd}^2$ ενώ η P_{stat} είναι ένας στατικός όρος που επηρεάζεται κυρίως από το ρεύμα διαρροής των τρανζίστορ. Η παράμετρος που αντιπροσωπεύει τη δραστηριότητα του κυκλώματος α, επηρεάζεται από τον αριθμό των bit που χρησιμοποιούνται στους υπολογισμούς όπως φαίνεται στον πίνακα I. Ο αριθμός των υπολογισμών σε ένα ευθύ πέρασμα ενός LSTM παρουσιάζεται στον πίνακα II. Η ενεργειακή μοντελοποίηση που πραγματοποιούμε δεν συμπεριλαμβάνει την ενέργεια που σπαταλάται σε υπολογισμούς που εξαρτώνται από την αρχιτεκτονική, και μπορεί να θεωρηθεί ως η ενέργεια που ξοδεύεται για όλους τους αριθμητικούς υπολογισμούς των LSMT (πολλαπλασιασμοί πίνακα-διανύσματος, διανυσματικοί πολλαπλασιασμοί και προσθέσεις στοιχείο προς στοιχείο).

	$\times$	$+$	Reg.	Wire	SRAM
α	$\mathcal{O}(n^2)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(n_{max})$

Πίνακας I: Δραστηριότητα κυκλώματος και στοιχειώδεις πράξεις.

Θα πρέπει να αναφερθεί πως οι αν και οι τεχνικές που παρουσιάζονται εδώ μπορούν να χρησιμοποιηθούν για την μείωση της ενεργειακής κατανάλωσης των νευρωνικών δικτύων κατά τη διαδικασία του ευθέος περάσματος, δεν επιβάλλουν κανέναν περιορισμό στην εκπαίδευση του δικτύου, η οποία μπορεί να πραγματοποιηθεί σε οποιοδήποτε υπολογιστικό περιβάλλον, με οποιοδήποτε σύνολο πόρων ή εργαλείων. Αυτό ισχύει γιατί ο πρωταρχικός μας στόχος

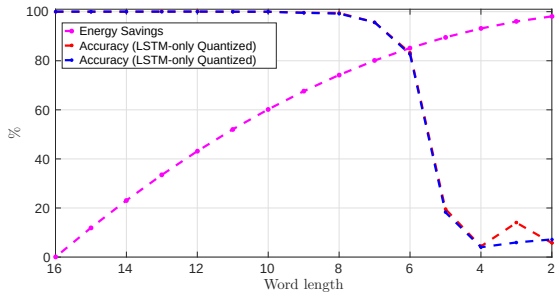
δεν είναι η ανάπτυξη αλλά η χρήση νευρωνικών δικτύων σε ενσωματωμένα συστήματα. Μια μελέτη που διερευνά τις συνέπειες της περιορισμένης αριθμητικής ακρίβειας κατά τη διαδικασία της εκπαίδευσης νευρωνικών δικτύων μπορεί να βρεθεί στο [15].

Α΄. Κβαντισμός

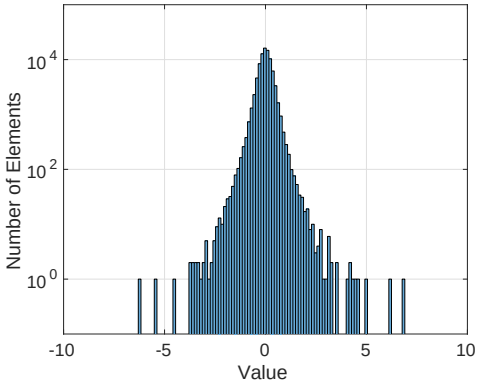
Σε σύγχρονα υπολογιστικά συστήματα, τα LSTM και άλλες νευρωνικές αρχιτεκτονικές χρησιμοποιούν αριθμητική κινητής υποδιαστολής 32 ή ακόμη και 64 bit. Οι ενεργειακές και χρονικές απαιτήσεις των πράξεων με τέτοιους αριθμούς, όπως και η πολυπλοκότητα των κυκλωμάτων που απαιτούνται, είναι ευρέως γνωστές [16] [17]. Αυτές οι αναπαραστάσεις μπορεί να είναι οι βέλτιστες από άποψη επιδόσεων σε γενικής χρήσης CPU ή GPU, αλλά το μεγάλο τους δυναμικό εύρος και η υψηλή τους ακρίβεια δεν συνεπάγονται υποχρεωτικά βελτίωση στην *ακρίβεια των προβλέψεων των νευρωνικών δικτύων*.

Όπως αναφέρθηκε στον πίνακα I, σημαντική εξοικονόμηση μπορεί να επιτευχθεί με τη μείωση του αριθμού των bit που χρησιμοποιούνται στους υπολογισμούς(π.χ. σε πολλαπλασιαστές, η μείωση αυτή μπορεί να επιφέρει $12\times$ μείωση στην ενέργεια με εισαγωγή 1%RMSE [18]). Για το λόγο αυτό και προκειμένου να ελαττώσουμε την κατανάλωση ενέργειας στου υπολογισμούς του LSTM, κβαντίζουμε τους πίνακες βαρών της αρχιτεκτονικής και τα δεδομένα εισόδου, χρησιμοποιώντας μια αναπαράσταση σταθερής υποδιαστολής με μήκος λέξης WL. Ο στόχος είναι να μελετηθεί η ελάττωση της ακρίβειας του δικτύου που οφείλεται στην μείωση της ανάλυσης των βαρών και των δεδομένων εισόδου. Προκειμένου να επιτευχθούν τα βέλτιστα αποτελέσματα, είναι σημαντικό να ληφθεί υπόψιν η στατιστική κατανομή των στοιχείων των πινάκων βαρών πριν τον κβαντισμό. Καθώς βάρη με μεγάλη απόλυτη τιμή υποδηλώνουν ισχυρές συνάψεις, είναι καλό να πραγματοποιηθεί κβαντισμός ο οποίος αποφεύγει τον κορεσμό των τιμών των μεταβλητών. Επίσης, η ακρίβεια σε στοιχεία με απόλυτη τιμή κοντά στο μηδέν θεωρείται λιγότερο σημαντική. Υπό αυτές τις υποθέσεις, μετασχηματίζουμε όλες τις παραμέτρους και τα δεδομένα εισόδου έτσι ώστε τα όρια της ζώνης κβαντισμού να συμπίπτουν με τις μέγιστες/ελάχιστες τιμές των παραμέτρων ή των δεδομένων εισόδου. Αυτό διασφαλίζει τη χρυσή τομή, εξασφαλίζοντας τόσο το απαραίτητο δυναμικό εύρος όσο και την ελάχιστη δυνατή απώλεια αριθμητικής ακρίβειας.

Σύμφωνα με την παραπάνω μέθοδο, κβαντίζουμε τα $\mathbf{W}_\star$, $\mathbf{R}_\star$, $\mathbf{b}_\star$ και για τα δύο LSTM. Αναφέρουμε πως η μετάβαση από αριθμητική ακρίβεια 64-bit κινητής υποδιαστολής σε 32-bit κινητής υποδιαστολής δεν επιφέρει καμία αλλαγή στην επίδοση του δικτύου, ενώ η μετάβαση σε αριθμητική σταθερής υποδιαστολής επιτρέπει τη χρήση μηκών λέξης μέχρι και $WL = 8$, χωρίς καμία απώλεια στην σχετική ακρίβεια των προβλέψεων του νευρωνικού δικτύου. Επιτρέποντας τη μείωση της σχετικής ακρίβειας σε 96%, ακόμη και 7 bit μπορούν να χρησιμοποιηθούν. Βασιζόμενοι στους αριθμούς των υπολογισμών που απαιτούνται για το ευθύ πέρασμα των LSTM που παρουσιάστηκαν στον πίνακα II,



Σχήμα 3: Η επίδραση του κβαντισμού στην ακρίβεια του LSTM και στην εξοικονόμηση ενέργειας. Περνώντας από 64 σε 32-bit αριθμητική κινητής υποδιαστολής δεν υπάρχει μεταβολή στην ακρίβεια. Κβαντίζοντας σε 8-bit αριθμητική σταθερής υποδιαστολής δεν υπάρχει μεταβολή της ακρίβειας, ενώ κβαντίζοντας σε 7-bit η σχετική ακρίβεια μειώνεται στο 96%. Ένα δίκτυο LSTM στα 7-bit καταναλώνει 80% λιγότερη ενέργεια από ένα αντίστοιχο 16-bit.



Σχήμα 4: Στατιστική κατανομή των παραμέτρων του προς-τα-εμπρός LSTM. Η μεγάλη πλειοψηφία των παραμέτρων έχει απόλυτη τιμή πολύ κοντά στο μηδέν, αλλά υπάρχουν εξωκείμενες τιμές που απέχουν μέχρι και 17σ από τη μέση τιμή. Το προς-τα-πίσω LSTM και το πλήρως διασυνδεδεμένο επίπεδο παρουσιάζουν παρόμοιες κατανομές. Προσέξτε τη λογαριθμική κλίμακα του άξονα y.

δείχνουμε πως μεταβαίνοντας από 16-bit σταθερής υποδιαστολής σε ακρίβεια 8 ή 7 bit, μπορούμε να επιτύχουμε μέχρι 74% και 80% εξοικονόμηση ενέργειας αντίστοιχα. Τα αποτελέσματα φαίνονται στο σχήμα 3.

Β΄. Κλάδεμα Νευρωνικών Συνάψεων

Όπως έχει παρατηρηθεί στο [19], τα νευρωνικά δίκτυα τείνουν να είναι σημαντικώς υπερπαραμετροποιημένα μοντέλα, ενώ διάφορες στρατηγικές έχουν αναπτυχθεί για την μείωση του αριθμού των βαρών των δικτύων. Καθώς οι σύγχρονες νευρωνικές αρχιτεκτονικές τείνουν να έχουν αριθμό παραμέτρων που εύκολα είναι της τάξης των εκατομμυρίων, είναι προφανές ότι οι υπολογισμοί που απαιτούνται έχουν επίσης και μεγάλες απαιτήσεις μνήμης και I/O. Η

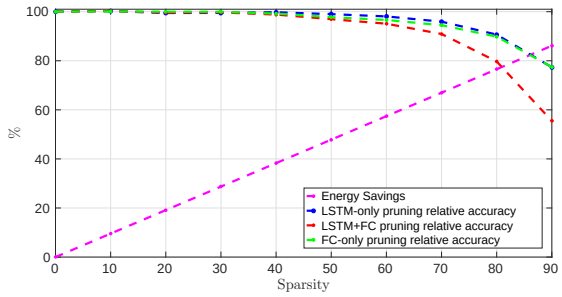
στρατηγική μας εδώ είναι να μελετήσουμε πως το κλάδεμα νευρωνικών συνάψεων επηρεάζει τις επιδόσεις των LSTM στις προβλέψεις τους και επίσης να μοντελοποιήσουμε τη επίδραση θα έχει το κλάδεμα αυτό στην ενεργειακή τους κατανάλωση.

Το σχήμα 4 δείχνει την κατανομή των παραμέτρων του προς-τα-μπροστά LSTM. Η κατανομή είναι σχετικά στενή και γύρω από το μηδέν αλλά έχει έναν μεγάλο αριθμό από εξωκείμενες τιμές, ακόμα και 17σ μακριά από τη μέση τιμή. Διαισθητικά μπορεί κανείς να καταλάβει ότι όσο μεγαλύτερη είναι η απόλυτη τιμή ενός βάρους, τόσο ισχυρότερη είναι η σύναψη μεταξύ δύο νευρώνων και ακολούθως, τόσο μεγαλύτερη είναι η σημασία της σύναψης αυτής για την εξαγωγή ενός συμπεράσματος από το νευρωνικό δίκτυο. Από στατιστική σκοπιά, αυτό υποδηλώνει πως η λειτουργία και η ακρίβεια του δικτύου εξαρτάται πρωτίστως από αυτές τις εξωκείμενες τιμές της στατιστικής κατανομής.

Για να μοντελοποιηθεί η διαδικασία του κλαδέματος νευρωνικών συνάψεων, όλες οι συνάψεις που χαρακτηρίζονται από βάρος με απόλυτη τιμή μικρότερη από ένα κατώφλι θεωρούνται περιττές και το βάρος τους τίθεται ίσο με μηδέν. Αυτή η διαδικασία ισοδυναμεί με την τεχνητή αύξηση του κατά πόσο ο πίνακας είναι αραιός (sparsity), και εννοιολογικά ισοδυναμεί με την αποσύζευξη μεταξύ των βασικών στοιχείων του συστήματος, των νευρώνων.

Καθώς κάθε πίνακας έχει τα δικά του στατιστικά γνωρίσματα, δεν θα ήταν "δίκαιο" να κλαδέψουμε όλες τις συνάψεις κάτω από ένα και μόνο κατώφλι. Αντί αυτού, σ'αυτό το πείραμα, η "αραιότητα" των πινάκων θεωρείται η ανεξάρτητη μεταβλητή και το κατώφλι που απαιτείται για την επίτευξη αυτής υπολογίζεται ανά πίνακα. Βάσει αυτής της κατωφλίωσης, όλα τα στοιχεία που έχουν απόλυτη τιμή μικρότερη από το κατώφλι του πίνακα στον οποίο ανήκουν μηδενίζονται. Χρησιμοποιώντας αυτό το σκεπτικό, γίνεται ευκολότερη η ποσοτικοποίηση της επιρροής που έχει το κλάδεμα στην ακρίβεια των προβλέψεων των LSTM. Δείχνουμε πως με 40% των συνάψεων να έχουν κλαδευτεί δεν υπάρχει καμία μείωση στην ακρίβεια, ενώ κλαδεύοντας 70% των συνάψεων η σχετική ακρίβεια ελαττώνεται στο 96%. Αξίζει να αναφερθεί πως το κλάδεμα του πλήρως διασυνδεδεμένου επιπέδου επιφέρει μεγαλύτερη μείωση στην ακρίβεια, καθώς έχει εκπαιδευτεί για να αναγνωρίζει γνωρίσματα των δεδομένων εισόδου σε υψηλότερο επίπεδο αφαιρετικότητας από τα επίπεδα LSTM που βρίσκονται κοντά στην είσοδο. Ευτυχώς, ο αριθμός των παραμέτρων του επιπέδου εξόδου είναι σχετικά μικρός σε σύγκριση με τον αριθμό των παραμέτρων όλης της αρχιτεκτονικής (μόνο 8% του συνολικού αριθμού οφείλεται στο 256×61 επίπεδο) και επομένως, μπορεί να αποφευχθεί το κλάδεμά του, ή να επιβληθεί ένα *ελαφρύτερο* κλάδεμα.

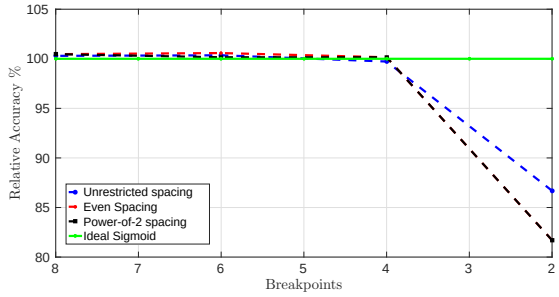
Από τη σκοπιά της ενεργειακής κατανάλωσης, διερευνούμε πως η *παράλειψη* μη απαραίτητων υπολογισμών επηρεάζει την εξοικονόμηση ενέργειας. Η παράλειψη υπολογισμών είναι περίπλοκο να επιτευχθεί σε επίπεδο λογισμικού, καθώς οι επιπλέον εντολές έλεγχου που θα ήταν απαραίτητες για να αποφασιστεί αν ένας υπολογισμός πρέπει να παραλειφθεί θα αναιρούσε την συνολική προσπάθεια για ελαχιστοποίηση της απαιτούμενης ενέργειας. Όμως, καθώς οι παράμετροι του δικτύου είναι γνωστές και σταθερές, τα βάρη των μη σημαντικών συνάψεων μπορούν να μηδενιστούν και έτσι ο έλεγχος μπορεί να υλοποιηθεί σε επίπεδο υλικού και να εξουδετερώσει την δραστηριότητα των πολλαπλασιαστών και των αθροιστών όταν το αποτέλεσμα μιας πράξης είναι γνωστό εκ των προτέρων. Βασιζόμενοι στον πίνακα II, αναγνωρίζουμε τι ποσοστό της αριθμητικής των LSTM εμπλέκει παραμέτρους γνωστές εκ των προτέρων και για το λόγο αυτό μπορεί να παραλειφθεί αποτελεσματικά. Για το δίκτυο υπό μελέτη (με διαστάσεις εισόδου $d = 39$, μέγεθος LSTM $h = 128$ και διαστάσεις εξόδου $l = 61$), πάνω από 97% των υπολογισμών προκύπτει από πολλαπλασιασμούς πινάκων-διανυσμάτων με γνωστές παραμέτρους. Με βάση αυτό, δείχνουμε πως δίκτυα κλαδεμένα κατά 40% και 70% χρησιμοποιούν 38% και 67% λιγότερη ενέργεια από ένα μη κλαδεμένο δίκτυο. Τα αποτελέσματα φαίνονται στο σχήμα 5.



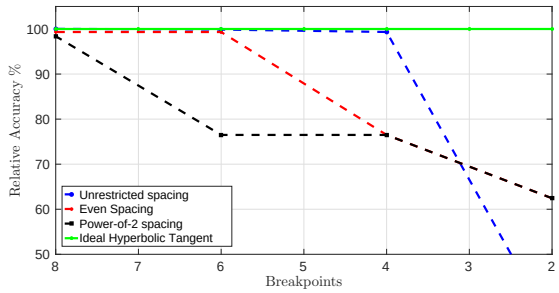
Σχήμα 5: Κλάδεμα νευρωνικών συνάψεων και κέρδος στην ενεργειακή κατανάλωση. Κλάδεμα μέχρι και 40% των συνάψεων των LSTM δεν επιφέρει καμία μεταβολή στην ακρίβεια των δικτύων και στη συνέχεια η ακρίβεια μειώνεται αργά, επιτρέποντας κλάδεμα 70% με αντίστοιχη ακρίβεια 96.1% σε σχέση με την ακρίβεια αναφοράς. Κλάδεμα μόνο του πλήρως διασυνδεδεμένου δικτύου οδηγεί σε παρόμοια συμπεριφορά, αλλά κλάδεμα και των δύο επιπέδων οδηγεί σε 95% σχετική ακρίβεια σε κλάδεμα 60% και ύστερα επιφέρει σημαντική επιδείνωση. Η εξοικονόμηση ενέργειας φαίνεται με μωβ.

Γ΄. Υλοποιήσεις μη γραμμικών συναρτήσεων ενεργοποίησης

Στον πίνακα II ο αριθμός των πολλαπλασιασμών και των προσθέσεων που απαιτούνται για την εφαρμογή μη γραμμικών συναρτήσεων ενεργοποίησης μοντελοποιούνται από τους συντελεστές σ_1, t_1 και σ_2, t_2 αντίστοιχα. Στη μελέτη αυτή υλοποιούμε εκδοχές των συναρτήσεων αυτών που βασίζονται σε πίνακες αναζήτησης (lookup tables, LUT), μεταβάλλοντας το πλήθος και την απόσταση μεταξύ των σημείων παρεμβολής. Η βέλτιστη λύση στο χώρο ενέργειας/ακρίβειας επιτυγχάνεται με τη χρήση μιας υλοποίησης *δύναμης-του-δύο*. Υπολογίζοντας τις μη γραμμικές συναρτήσεις σε σημεία που είναι δυνάμεις του 2, μπορεί να χρησιμοποιηθεί ένας πολύ απλός μηχανισμός παρεμβολής που απαιτεί μία



(α΄) Η συνάρτηση ενεργοποίησης $\sigma(x)$



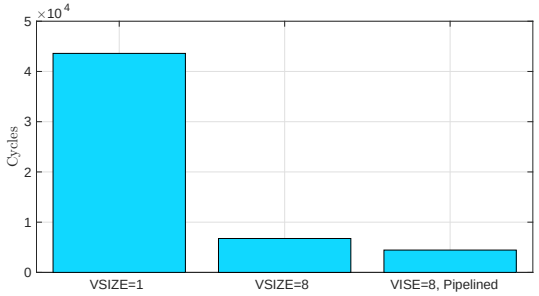
(β΄) Η συνάρτηση ενεργοποίησης $\tanh(x)$

Σχήμα 6: Η επίδραση των προσεγγίσεων των συναρτήσεων ενεργοποίησης στην ακρίβεια. Οι τρεις περιπτώσεις που εξετάζονται ειναι οι *ελεύθερη τοποθέτηση σημείων παρεμβολής* (μπλε), *τοποθέτηση σημείων παρεμβολής σε τακτά διαστήματα* (κόκκινο), *τοποθέτηση σημείων σε δυνάμεις του δύο* (μαύρο). Στο (a) δείχνεται ότι ακόμα και αν η σιγμοειδής συνάρτηση αντικατασταθεί με την φθηνότερη υπολογιστικά προσέγγιση *δύναμης-του-δύο* χρησιμοποιώντας 4 σημεία παρεμβολής, δεν υπεισέρχεται καμία επιδείνωση στην ακρίβεια. Στο (b), η προσέγγιση της υπερβολικής εφαπτομένης $\tanh(x)$ με την ίδια στρατηγική απαιτεί 8 σημεία για να επιτύχει τέλεια σχετική ακρίβεια.

ολίσθηση, την εφαρμογή μιας μάσκας από bit, έναν πολλαπλασιασμό ($\sigma_1 = t_1 = 1$) και δύο προσθέσεις ($\sigma_2 = t_2 = 2$). Επίσης, αξιολογήθηκαν και άλλες μέθοδοι που απαιτούν λιγότερα σημεία παρεμβολής αλλά πιο περίπλοκους μηχανισμούς παρεμβολής. Το σχήμα 6 δείχνει ότι ακόμα και με τις απλούστερες των προσεγγίσεων (δύναμη-του-δύο) και χρησιμοποιώντας 8 σημεία παρεμβολής, καμία μείωση δεν επέρχεται στην δυνατότητα του νευρωνικού δικτύου να κάνει σωστές προβλέψεις για τα εισερχόμενα δείγματα ομιλίας. Θεωρούμε πως αυτό το εύρημα είναι ουσιαστικής σημασίας, καθώς δείχνει πως η πολύπλοκη δομή των LSTM μπορεί να λειτουργήσει αποτελεσματικά ακόμη και όταν οι χρησιμοποιούμενες συναρτήσεις ενεργοποίησης απέχουν πολύ από τις ιδανικές.

V. Υλοποίηση σε υλικό

Στην ενότητα IV δείξαμε ότι οι επιδόσεις των LSTM δεν υφίστανται σημαντική βλάβη, ακόμη και όταν τα υπολογι-



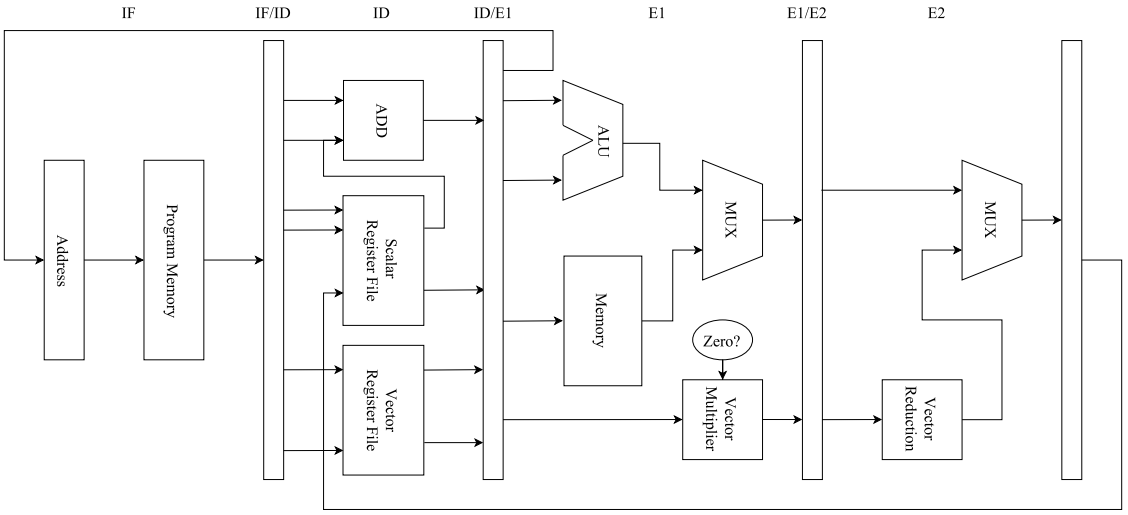
Σχήμα 7: Η εξοικονόμηση σε κύκλους μηχανής που μπορεί να επιτευχθεί για έναν πολλαπλασιασμό πίνακα-διανύσματος διαστάσεων $(48 \times 128) \times (128 \times 1)$ χρησιμοποιώντας SIMD και διοχέτευση σε σχέση με μια αρχιτεκτονική Single Instruction Single Data (SISD).

στικά τους στοιχεία και οι παράμετροί τους αντικαταστα-θούν με πολύ απλοποιημένες προσεγγίσεις αυτών. Εδώ παρουσιάζουμε μια αρχιτεκτονική υλικού που εκμεταλλεύεται την προαναφερθείσα ιδιότητα, χρησιμοποιώντας έναν επεξεργαστή ειδικής χρήσης με μεταβλητό μήκος λέξης.

Το μειωμένο μήκος λέξης της διαδρομής δεδομένων κάνει πραγματικότητα την εξοικονόμηση ενέργειας που παρουσιάστηκε στην ενότητα IV- . Μία διανυσματική μονάδα επεξεργασίας (vector processing unit, VPU) χρησιμοποιείται για την εκμετάλλευση του παραλληλισμού που εγγενώς υπάρχει στους υπολογισμούς των νευρωνικών δικτύων, επιταχύνοντας τους πολλαπλασιασμούς πινάκων-διανυσμάτων, εκτελώντας παράλληλα διανυσματικούς πολλαπλασιασμούς και προσθέσεις στοιχείο προς στοιχείο και εφαρμόζοντας παράλληλα τις μη γραμμικές συναρτήσεις ενεργοποίησης.

Ακόμη, επεκτείνουμε το σύνολο εντολών (instruction set) του επεξεργαστή με εντολές που επιτρέπουν την φόρτωση δύο διανυσμάτων από τη μνήμη, τον στοιχείο-με-στοιχείο πολλαπλασιασμό τους και την άθροιση των μερικών αθροισμάτων σε έναν κύκλο μηχανής. Για να το επιτύχουμε αυτό, τροποποιούμε τις μεθόδους πρόσβασης μνήμης (memory accessing) και ορίζουμε νέα στάδια στη διοχέτευση (pipelining). Ο τροποποιημένος επεξεργαστής φαίνεται στο σχήμα 8.

Η χρήση εντολών SIMD μέσω της διανυσματικής μονάδας επεξεργασίας και ο ορισμός επιπλέον σταδίων στη διοχέτευση ελαττώνουν τον συνολικό αριθμό κύκλων μηχανής που απαιτούνται για το ευθύ πέρασμα του LSTM. Αυτό επιτρέπει την εφαρμογή τεχνικών κλιμάκωσης συχνότητας και τάσης (Dynamic Frequency and Voltage Scaling) για ακόμη μεγαλύτερη εξοικονόμηση ενέργειας. Για να εκμεταλλευτούμε αποτελεσματικά την εξειδικευμένη αρχιτεκτονική αυτή, οι βασικοί υπολογιστικοί πυρήνες των LSTM επαναϋλοποιήθηκαν σε C. Το κέρδος σε κύκλους μηχανής κατά τους πολλαπλασιασμούς πίνακα-διανύσματος (που σύμφωνα με την ανάλυση της ενότητας IV- είναι υπαίτιοι για 97% όλων των υπολογισμών κατά το ευθύ πέρασμα) φαίνεται στο σχήμα 7. Πέρα από τα κέρδη που εκφράζονται σε κύκλους και που



Σχήμα 8: Η τροποποιημένη αρχιτεκτονική. Η αρχιτεκτονική αυτή έχει τις εξής τροποποιήσεις: α) Χρησιμοποιεί διανυσματικούς καταχωρητές και έναν διανυσματικό πολλαπλασιαστή ο οποίος ελέγχει σε επίπεδο υλικού αν κάποια από τις εισόδους του είναι μηδέν, ώστε να αποφύγει αχρείαστη δραστηριότητα κυκλώματος και β) Διαθέτει ένα επιπλέον στάδιο διοχέτευσης E2 όπου τα μερικά γινόμενα αθροίζονται πριν γραφτούν πίσω στους διανυσματικού καταχωρητές. Με αυτή τη δομή, ένα εσωτερικό γινόμενο δύο διανυσμάτων μπορεί να υπολογιστή σε ακριβώς έναν κύκλο μηχανής, επιτρέποντας την επιτάχυνση του πολλαπλασιασμού πίνακα-διανύσματος.

μπορούν να προσομοιωθούν με προσομοιωτές συνόλου εντολών (instruction set simulators) η περιγραφή του κυκλώματος της διανυσματικής μονάδας επεξεργασίας είναι τέτοια ώστε να εφαρμόζει έλεγχο για μηδενικές τιμές σε επίπεδο κυκλώματος. Παρά το γεγονός ότι αυτό δε μπορεί να μοντελοποιηθεί σε επίπεδο κύκλων, οδηγεί σε σημαντική εξοικονόμηση ενέργειας όταν εφαρμόζεται κλάδεμα (όπως περιγράφεται στην ενότητα IV-), καθώς ελαττώνει τη δραστηριότητα των κυκλωμάτων του διανυσματικού πολλαπλασιαστή.

VI. Σύνοψη

Σε αυτή την μελέτη, αναπτύσσουμε μια νευρωνική αρχιτεκτονική που βασίζεται στη χρήση LSTM νευρωνικών δικτύων και την αξιολογούμε πάνω σε ένα ακολουθιακό πρόβλημα μάθησης μηχανών, την Αυτόματη Αναγνώριση Ομιλίας, χρησιμοποιώντας το σύνολο δεδομένων TIMIT. Μετά την απόκτηση ορισμένων δεδομένων για τις επιδόσεις του συστήματος, διερευνούμε τις απώλειες ακρίβειας που προκύπτουν όταν προσεγγιστικές μέθοδοι επεξεργασίας δεδομένων εφαρμόζονται στο ευθύ πέρασμα της αρχιτεκτονικής. Αυτές οι μέθοδοι συμπεριλαμβάνουν τον κβαντισμό παραμέτρων και δεδομένων σε μικρά μήκη λέξης, το κλάδεμα νευρωνικών συνάψεων στα LSTM και την αντικατάσταση των μη γραμμικών συναρτήσεων ενεργοποίησης με απλουστευμένες προσεγγίσεις τους. Το κίνητρο πίσω από αυτή την αναζήτηση είναι το γεγονός πως η κατά-προσέγγιση επεξεργασία δεδομένων μπορεί να οδηγήσει σε σημαντική εξοικονόμηση ενέργειας, πράγμα που θα διευκολύνει, με τη σειρά του, την υλοποίηση λύσεων που εκμεταλλεύονται την μηχανική μάθηση σε ενσωματωμένα συστήματα τα οποία πρέπει να έχουν, εν γένει, πολύ περιορισμένες ενεργειακές απαιτήσεις.

Δείχνουμε ότι καμία υποβάθμιση δεν υπεισέρχεται στην ακρίβεια των προβλέψεων των LSTM όταν χρησιμοποιείται μέχρι και 8 bit ακρίβεια κατά τους υπολογισμούς τους αντί των συνηθέστερων αναπαραστάσεων των 64 και 32 bits, ότι κλαδεύοντας μέχρι και 70% των νευρωνικών συνάψεων οδηγεί σε λιγότερο από 4% μείωση στην σχετική τους ακρίβεια και ότι υλοποιήσεις των μη γραμμικών συναρτήσεων ενεργοποίησης που χρησιμοποιούν πίνακες αναζήτησης με μέχρι και 8 σημεία παρεμβολής δεν επιφέρουν καμία βλάβη στην ακρίβεια των δικτύων. Εκτιμούμε 80% εξοικονόμηση ενέργειας (με κβαντισμό στα 7 bit) και 67% εξοικονόμηση (με κλάδεμα 70%). Τελικά, προτείνουμε μια αρχιτεκτονική υλικού που μπορεί να εκμεταλλευτεί αυτές τις προσεγγιστικές τεχνικές προκειμένου να επιτύχει τις ενεργειακές αυτές βελτιώσεις.

Αναφορές

[1] A. Graves and N. Jaitly, "Towards End-To-End Speech Recognition with Recurrent Neural Networks," *JMLR Workshop and Conference Proceedings*, vol. 32, no. 1, pp. 1764–1772, 2014. [Online]. Available: http://jmlr.org/proceedings/papers/v32/graves14.pdf

[2] 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.

[3] J. Donahue, L. Anne Hendricks, S. Guadarrama, M. Rohrbach, S. Venugopalan, K. Saenko, and T. Darrell, "Long-term recurrent convolutional networks for visual recognition and description," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2015, pp. 2625–2634.

[4] N. Srivastava, E. Mansimov, and R. Salakhutdinov, "Unsupervised learning of video representations using lstms," *arXiv preprint arXiv:1502.04681*, 2015.

[5] A. Graves, “Generating sequences with recurrent neural networks,” *arXiv preprint arXiv:1308.0850*, pp. 1–43, 2013. [Online]. Available: http://arxiv.org/abs/1308.0850

[6] S. K. Sønderby and O. Winther, “Protein secondary structure prediction with long short term memory networks,” *arXiv preprint arXiv:1412.7828*, 2014.

[7] M. Jiang and G. Gielen, “The effects of quantization on multi-layer feedforward neural networks,” *International journal of pattern recognition and artificial intelligence*, vol. 17, no. 04, pp. 637–661, 2003.

[8] G. Dundar and K. Rose, “The effects of quantization on multilayer neural networks.” *IEEE transactions on neural networks/a publication of the IEEE Neural Networks Council*, vol. 6, no. 6, pp. 1446–1451, 1994.

[9] Z. Du, A. Lingamneni, Y. Chen, K. Palem, O. Temam, and C. Wu, “Leveraging the Error Resilience of Neural Networks for Designing Highly Energy Efficient Accelerators,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 34, no. 8, pp. 1223–1235, 2015.

[10] J. S. Garofolo, L. F. Lamel, W. M. Fisher, J. G. Fiscus, and D. S. Pallett, “DARPA TIMIT acoustic-phonetic continous speech corpus CD-ROM. NIST speech disc 1-1.1,” *NASA STI/Recon Technical Report N*, vol. 93, Feb. 1993.

[11] C. Lopes and F. Perdigão, “Phone recognition on the TIMIT database,” *Speech Technologies/Book*, vol. 1, pp. 285–302, 2011.

[12] S. Seneff and V. Zue, “Transcription and alignment of the timit database,” *TIMIT CD-ROM Documentation*, 1988.

[13] R. Collobert, S. Bengio, and J. Mariéthoz, “Torch: a modular machine learning software library,” IDIAP, Tech. Rep., 2002.

[14] P. J. Werbos, “Backpropagation through time: what it does and how to do it,” *Proceedings of the IEEE*, vol. 78, no. 10, pp. 1550–1560, 1990.

[15] S. Gupta, A. Agrawal, K. Gopalakrishnan, and P. Narayanan, “Deep Learning with Limited Numerical Precision,” *arXiv:1502.02551 [cs, stat]*, vol. 37, 2015. [Online]. Available: http://arxiv.org/abs/1502.02551$\delimiter”026E30F$nhttp://www.arxiv.org/pdf/1502.02551.pdf

[16] J. L. Hennessy and D. A. Patterson, *Computer architecture: a quantitative approach*. Elsevier, 2011.

[17] G. Govindu, L. Zhuo, S. Choi, and V. Prasanna, “Analysis of high-performance floating-point arithmetic on fpgas,” in *Parallel and Distributed Processing Symposium, 2004. Proceedings. 18th International*. IEEE, 2004, p. 149.

[18] B. Moons, B. De Brabandere, L. Van Gool, and M. Verhelst, “Energy-efficient convnets through approximate computing,” *arXiv preprint arXiv:1603.06777*, 2016.

[19] J. Pool and W. J. Dally, “Neural Networks,” pp. 1–9.

Bibliography

[1] O. Abdel-hamid, L. Deng, and D. Yu. Exploring Convolutional Neural Network Structures and Optimization Techniques for Speech Recognition. *14th Annual Conference of the International Speech Communication Association, (INTERSPEECH 2013)*, (August):3366–3370, 2013.

[2] Acoustical Society of America. *ANSI/ASA Standard S1.1-2013*, 2013.

[3] W. Byeon, M. Liwicki, and T. M. Breuel. Scene analysis by mid-level attribute learning using 2d lstm networks and an application to web-image tagging. *Pattern Recognition Letters*, 63:23–29, 2015.

[4] R. Collobert, S. Bengio, and J. Mariéthoz. Torch: a modular machine learning software library. Technical report, IDIAP, 2002.

[5] R. Collobert, K. Kavukcuoglu, and C. Farabet. Implementing neural networks efficiently. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 7700 LECTU:537–557, 2012.

[6] D. Crystal. *The Cambridge encyclopedia of language*, volume 2. Cambridge Univ. Press.

[7] K. H. Davis, R. Biddulph, and S. Balashek. Automatic recognition of spoken digits. *The Journal Of The Acoustic Society Of America*, 24(6):637 – 642, 1952.

[8] R. Drullman. Temporal envelope and fine structure cues for speech intelligibility. *Journal of the Acoustical Society of America*, 97(1):585–592, 1995.

[9] J. S. Garofolo, L. F. Lamel, W. M. Fisher, J. G. Fiscus, and D. S. Pallett. DARPA TIMIT acoustic-phonetic continous speech corpus CD-ROM. NIST speech disc 1-1.1. *NASA STI/Recon Technical Report N*, 93, Feb. 1993.

[10] U. G. Goldstein. An articulatory model for the vocal tracts of growing children. 1977.

[11] A. Graves. *Supervised sequence labelling*. Springer, 2012.

[12] A. Graves and N. Jaitly. Towards End-To-End Speech Recognition with Recurrent Neural Networks. *JMLR Workshop and Conference Proceedings*, 32(1):1764–1772, 2014.

[13] A. Graves, A. Mohamed, and G. Hinton. Speech Recognition with Deep Recurrent Neural Networks. *IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, (3):6645–6649, 2013.

[14] A. Graves and J. Schmidhuber. Framewise phoneme classification with bidirectional lstm and other neural network architectures. *Neural Networks*, 18(5):602–610, 2005.

[15] K. Greff, R. K. Srivastava, J. Koutník, B. R. Steunebrink, and J. Schmidhuber. LSTM: A Search Space Odyssey. *arXiv*, page 10, 2015.

[16] A. K. Halberstadt. *Heterogeneous acoustic measurements and multiple classifiers for speech recognition.* PhD thesis, Massachusetts Institute of Technology, 1998.

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

[18] M. Horowitz. Energy table for 45nm process. Stanford VLSI wiki.

[19] C. Ittichaichareon, S. Suksri, and T. Yingthawornsuk. Speech recognition using mfcc. In *International Conference on Computer Graphics, Simulation and Modeling (ICGSM'2012) July*, pages 28–29, 2012.

[20] B. H. Juang and L. R. Rabiner. Automatic Speech Recognition - A Brief History of the Technology Development. *Elsevier Encyclopedia of Language and Linguistics*, pages 1–24, 2005.

[21] Y. LeCun, L. Jackel, L. Bottou, a. Brunot, C. Cortes, J. Denker, H. Drucker, I. Guyon, U. Müller, E. Säckinger, P. Simard, and V. Vapnik. Comparison of learning algorithms for handwritten digit recognition. *International Conference on artificial neural networks*, pages 53–60, 1995.

[22] C. Lopes and F. Perdigão. Phone recognition on the TIMIT database. *Speech Technologies/Book*, 1:285–302, 2011.

[23] P. Mermelstein. Distance measures for speech recognition, psychological and instrumental. *Pattern recognition and artificial intelligence*, 116:374–388, 1976.

[24] A. Newell. HARPY: Production systems and human cognition. 1978.

[25] D. OShaughnessy. Invited paper: Automatic speech recognition: History, methods and challenges. *Pattern Recognition*, 41(10):2965–2979, 2008.

[26] S. B. Park, J. W. Lee, and S. K. Kim. Content-based image classification using a neural network. *Pattern Recognition Letters*, 25(3):287 – 300, 2004.

[27] R. Pascanu, T. Mikolov, and Y. Bengio. On the difficulty of training recurrent neural networks. *Proceedings of The 30th International Conference on Machine Learning*, (2):1310–1318, 2012.

[28] D. Poeppel and P. J. Monahan. Speech perception: Cognitive foundations and cortical implementation. *Current Directions in Psychological Science*, 17(2):80–85, 2008.

[29] T. Sakai and S. Doshita. An Automatic Recognition System of Speech Sounds.

[30] R. V. Shannon, F.-G. Zeng, V. Kamath, J. Wygonski, and M. Ekelid. Speech recognition with primarily temporal cues. *Science*, 270(5234):303–304, 1995.

[31] N. Srivastava, E. Mansimov, and R. Salakhutdinov. Unsupervised learning of video representations using lstms. *arXiv preprint arXiv:1502.04681*, 2015.

[32] J. Von Neumann. Probabilistic logics and the synthesis of reliable organisms from unreliable components. *Automata studies*, 34:43–98, 1956.

[33] L. Wanhammar. *DSP integrated circuits*. Academic press, 1999.

[34] P. J. Werbos. Backpropagation through time: what it does and how to do it. *Proceedings of the IEEE*, 78(10):1550–1560, 1990.

[35] M. Wöllmer, M. Kaiser, F. Eyben, B. Schuller, and G. Rigoll. LSTM-modeling of continuous emotions in an audiovisual affect recognition framework. *Image and Vision Computing*, 31(2):153–163, 2013.

[36] V. Zue, S. Seneff, and J. Glass. Speech database development at mit: Timit and beyond. *Speech Communication*, 9(4):351 – 356, 1990.

Master thesis filing card

Student: Georgios N. Evangelopoulos

Title: Efficient Hardware Mapping of Long Short-Term Memory Neural Networks for Automatic Speech Recognition

UDC: 621.3

Abstract:

Recently, Long Short-Term Memory Neural Networks (LSTMs) have become state-of-the-art models for a variety of machine learning tasks, including but not limited to Automatic Speech Recognition, video and action classification, unsegmented handwriting recognition and generation, polyphonic music modelling and protein secondary structure prediction. However, LSTMs are usually very memory and computationally intensive. In order to enable LSTM-based classification in embedded systems such as smartphones and ubiquitous electronics for wireless sensor networks and the internet of things, their energy footprint should be reduced. This work examines how LSTMs behave as different methods of approximate computing are applied. Using shorter word lengths to represent architectural parameters and data, pruning LSTM connections and coarsely approximating the non-linear activation functions present in neural networks are some of the techniques applied. These algorithmic-level techniques are combined with architectural modifications to an Application Specific Instruction Set Processor (ASIP) to reduce energy consumption in an LSTM-based Automatic Speech Recognition system. These modifications include parallelism in the processor data path, introduction of more pipeline stages, extensions to the Instruction Set Architecture and RTL-level modifications that can further reduce the switching activity of the processor. Our results show that LSTM neural networks allow for many algorithmic simplifications while suffering little to no penalty in their predictive performance. We show that LSTMs behave well even when undergoing the simplest approximations and that when combined with a hardware architecture that can realize the predicted energy savings, they can be efficiently used for embedded sequential machine learning tasks on platforms with a limited energy budget.

Thesis submitted for the degree of Master of Science in Electrical Engineering, option Electronics and Integrated Circuits

Thesis supervisor: Prof. dr. ir. Marian Verhelst

Assessors: Prof. dr. ir. W. Dehaene
Prof. dr. ir.H. van Hamme

Mentor: Ir. Bert Moons