UNIVERSIDADE ESTADUAL DE CAMPINAS
Faculdade de Engenharia Elétrica e de Computação

Mijail Ronald Vidal Vargas

Artificial Neural Networks in Embedded Systems Applications for User Authentication

Redes Neurais Artificiais em Aplicações de Sistemas Embarcados para Autenticação de Usuários

Campinas

2019

Mijail Ronald Vidal Vargas

Artificial Neural Networks in Embedded Systems Applications for User Authentication

Redes Neurais Artificiais em Aplicações de Sistemas Embarcados para Autenticação de Usuários

Dissertation presented to the School of Electrical and Computer Engineering of the University of Campinas in partial fulfillment of the requirements for the degree of Master in Electrical Engineering, in the area of Computer Engineering.

Dissertação apresentada à Faculdade de Engenharia Elétrica e de Computação da Universidade Estadual de Campinas como parte dos requisitos exigidos para a obtenção do título de Mestre em Engenharia Elétrica, na Área de Engenharia de Computação.

Supervisor: Prof. Dr. Eduardo Alves do Valle Júnior

Este trabalho corresponde à versão final da dissertação defendida pelo aluno Mijail Ronald Vidal Vargas, e orientada pelo Prof. Dr. Eduardo Alves do Valle Júnior.

Campinas

2019

Ficha catalográfica
Universidade Estadual de Campinas
Biblioteca da Área de Engenharia e Arquitetura
Rose Meire da Silva - CRB 8/5974

V667a

Vidal Vargas, Mijail Ronald, 1990-
Artificial neural networks in embedded systems applications for user authentication / Mijail Ronald Vidal Vargas. – Campinas, SP : [s.n.], 2019.

Orientador: Eduardo Alves do Valle Júnior.
Dissertação (mestrado) – Universidade Estadual de Campinas, Faculdade de Engenharia Elétrica e de Computação.

1. Redes Neurais (Computação). 2. FPGA (arranjo de Lógica Programável em Campo). 3. Sistemas embarcados (Computadores). I. Valle Júnior, Eduardo Alves do, 1978-. II. Universidade Estadual de Campinas. Faculdade de Engenharia Elétrica e de Computação. III. Título.

Informações para Biblioteca Digital

Título em outro idioma: Redes neurais artificiais em aplicações de sistemas embarcados para autenticação de usuários
Palavras-chave em inglês:
Neural networks
FPGA (Field programmable gate arrays)
Embedded systems
Área de concentração: Engenharia de Computação
Titulação: Mestre em Engenharia Elétrica
Banca examinadora:
Eduardo Alves do Valle Júnior [Orientador]
Leandro Tiago Manêra
Vanderlei Bonato
Data de defesa: 12-11-2019
Programa de Pós-Graduação: Engenharia Elétrica

Identificação e informações acadêmicas do(a) aluno(a)
- ORCID do autor: https://orcid.org/0000-0002-2870-6887
- Currículo Lattes do autor: http://lattes.cnpq.br/6082323296333956

COMISSÃO JULGADORA - DISSERTAÇÃO DE MESTRADO

Candidato: Mijail Ronald Vidal Vargas RA: 162644

Data de defesa: 12 de Novembro de 2019

Título da Dissertação: "Redes Neurais Artificiais em Aplicações de Sistemas Embarcados para autenticação de Usuários"

Prof. Dr. Eduardo Alves do Valle Júnior (Presidente)

Prof. Dr. Leandro Tiago Manêra

Dr. Vanderlei Bonato

A Ata de Defesa, com as respectivas assinaturas dos membros da Comissão Julgadora, encontra-se no SIGA (Sistema de Fluxo de Dissertaçãoes) e na Secretaria de Pós-Graduação da Faculdade de Engenharia Elétrica e de Computação.

Acknowledgements

First, I must thank God for having the health and strength to finish this work.

This work would not have been possible without Prof. Eduardo Valle, for his mentorship and guidance in this work, who gave me the opportunity, confidence, and constant support in this dissertation.

Moreover, to Prof. Anderson Rocha for all the support and guidance in developing the experiments.

My laboratory, RECOD, provided me with the computational resources necessary for this dissertation, and also a fantastic research environment with unique collaborations, opportunities, and discussions have taken place.

My parents, Humberto Vidal and Ana Alicia Vargas, for all their love and help during this process. My brothers, Cesar, Sergio, and Pablo, supported me constantly. Moreover, my sister Janeth always helped me in every phase of my life.

I thank FUCAMP and LG for the overall financial support of my research group and laboratory.

‘In the middle of difficulty lies opportunity’
-Albert Einstein

Resumo

Um dispositivo móvel é uma ferramenta essencial para o trabalho que vai substituindo um computador desktop em muitas atividades da vida diária, mas em caso de perda do despositivo, pode comprometer as informações pessoais.

Normalmente, um dispositivo móvel oferece apenas uma camada de segurança após o primeiro acesso, como ser palavras-chave, PINs e perguntas secretas, não sendo suficiente em um problema de autenticação contínua. É por isso que a autenticação contínua tem sido um tópico ativo nos últimos anos, e tem proposto muitas técnicas para monitorar o acesso aos usuários. A técnica mais comum é o reconhecimento de gestos, este metodo utiliza sequências de movimentos na tela sensível ao toque para autenticação do usuário. Além disso, através do uso de algoritmos de aprendizado de máquina, mostraram um avanço significativo na área de segurança para telefones celulares.

Neste trabalho, propusemos dois novos tipos de algoritmos de autenticação contínua usando deep learning: O primeiro baseado em software usando uma arquitetura convolucional (Cifar-10) como extrator de recursos e um classificador Support Vector Machine (SVM) para autenticação de usuários. Neste caso, os resultados foram ótimos para condições normais onde o usuário interage em um único aplicativo, mas para um aplicativo diferente, o sistema não se mostrou muito robusto para rejeitar pessoas falsas porque o esquema de gestos depende do aplicativo gerado. Além disso, seu uso exaustivo causa a alocação de uma grande quantidade de memória RAM, o que faz com que a bateria seja consumida rapidamente.

O segunda proposta foi baseado em hardware e é chamado Mini Mind. A arquitetura Mini Mind foi implementada em um Cyclone IV - FPGA com duas camadas de LSTM. Os resultados do Mini Mind podem computar um gesto 2,5 vezes mais rápido que a CPU e 14 vezes mais rápido que a GPU. Em termos de consumo de energia, pode consumir 17 vezes menos do que uma CPU e 109 vezes menos que uma GPU.

Ambas as arquiteturas, software e hardware diferem significativamente entre desenho e implementação. Mas ambos os pontos de vista são considerados em sistemas embarcados para autenticação contínua de usuários.

Palavras-chaves: Redes neurais; FPGA; Sistemas embarcados.

Abstract

A mobile device is an essential tool for working and also it is replacing a desktop computer in many activities of daily life, but in the case of lost, it could compromise the personal information. Typically, a mobile device offers only one security layer after first access such as keywords, PINs, and secret questions, being not enough in a continuous authentication problem. That is why continuous authentication has been an active topic in the last year, and it has proposed many techniques for monitoring all the time user access with the use of sensors in a mobile device. The most common technique is gesture recognition, which used touchscreen sequences movements for user authentication. Moreover, through the use of algorithms of machine learning, they showed a significant advance in the area of security for mobile phones.

In this work, we proposed two new types of continuous authentication algorithms using deep learning: Based on software using a convolutional architecture (Cifar-10) as a feature extractor and a Support Vector Machine (SVM) classifier for user authentication. In this case, the results were optimal for normal conditions where the user interacts reasonably in a single application, but for a different application, the system did not much robust to reject fake people because the gesture scheme depends on the application generated. Besides, its exhaustive use causes the allocation of a large amount of RAM, which causes the battery to be consumed quickly.

The second-based hardware is called Mini Mind. Mini Mind architecture was implemented in a Cyclone IV - FPGA with two layers of LSTM and fully connected. Also, it was provided to be efficient in terms of power consumption and throughput for a continuous authentication process. In this case, the results Mini Mind can compute one gesture 2.5 times faster than CPU and 14 times faster than GPU. In terms of power consumption, it can consume 17 times less than CPU and 109 times less than GPU.

Both architectures, software, and hardware significantly differ between design and implementation. But both points of view are considered in embedded systems for continuous user authentication.

Keywords: Neural Network; FPGA; Embedded Systems.

List of Figures

List of Tables

Contents

1 Introduction

For the last decade, artificial neural networks have promoted a striking rebirth of artificial intelligence. Dedicated hardware — and, in special, embedded hardware — for them has attracted increasing interest. Embedded devices have gained attention in the last year; applications such as image analysis, computer vision, and robotics have appeared. In this context, using these devices requires an extra layer of security than the traditional (e.g., Pin and screen pattern) to protect user data. That is why user authentication, which we will choose as the main testbed for the hardware developed in this work (Chapter 3), has attracted particular interest.

Artificial intelligence has been a very active topic in the last decade, with steep advances in the art for many applications such as computer vision, speech recognition, natural language processing, and support for many works (BENGIO; COURVILLE, 2016). *Deep learning*, introduced about ten years ago, is the main (but not the only) responsible for those advances. Deep learning models are "deep" artificial neural networks, i.e., they have a large number of hidden layers. They are often end-to-end models, which work seamlessly from raw data until decision output, without the need for handcrafted feature extraction — which was critical for previous models. By combining many elementary operations such as convolutions, dot products, and activation functions, those models can make complex decisions with high accuracy.

The sheer number of parameters and operations (in the order of tens of millions) makes deep learning models very computationally expensive in terms of space, time, and power consumption!, creating a significant challenge for deploying those models in embedded systems, which have stringent limitations of memory, processing, energy, or cost (or all at once). Implementing deep models in embedded hardware is very challenging and requires creative efforts to minimize models' requirements, preserve prediction accuracy, and maximize utilization of the limited hardware.

Consider user authentication in mobile devices — a market that is expected to reach nearly 3.5 billion devices at the end of 2020, i.e., approximately *half the global population* (2014, ; 2019, ; POPULATIONPYRAMID,). Today's cellphones recognize fingerprints and even individual faces — very complex tasks — which must be performed fast, dozens of times per day, without draining the device's battery. Mobile devices are essential for personal and professional life and provide access to very sensitive information such as e-mail, schedules, lists of contacts — and increasingly also allow making payments. Thus, authentication schemes must also be very reliable.

In terms of security, mobiles are not immune to attacks. For example, the screen lock type is vulnerable to smudge attacks, which use the residual oil left by the user's fingers on the device to make a replica that might fool the authenticator in a large number of cases (AVIV et al., 2010). In the case of biometrics authentication, such as fingerprint readers, iris, and face recognition, those offer only one security layer after the first access. In this context, *continuous authentication* was proposed to improve security by evaluating the entirety of the user interaction with the device instead of just the initial access. Biometric (or behavioral, or usage) patterns that deviate too from the past may reveal malicious access (PATEL et al., 2016). Those techniques employ various sensors in a mobile device– gyroscopes, accelerometer, orientation sensors, pressure sensors, touch screen– to create and validate the user's profile.

In this thesis, we will design a new architecture for deep learning from the ground up, aiming at very strict memory and processing constraints. We have chosen continuous authentication — in our case by using the user's gestures on the touchscreen — as a testbed application for our hardware application to showcase its equivalence to a solution implemented in software.

1.1 Motivation

Current technology faces a tension between artificial intelligence models that are becoming increasingly computational complex and the growing popularity of low-power computational devices brought by *ubiquitous computing* — the notion that *everything*, from an airplane to a light switch, should be integrated into a cooperating network of intelligent devices. As more and more objects around us become "smart", the challenge of bringing advanced intelligence to them grows.

One way to resolve that tension is by using connected networks and cloud computing (Figure 1.1). In that model, made possible, for example, by the so-called *Internet-of-Things*, smart devices have sensors and actuators but do not necessarily have high computational capabilities: complex decisions are delegated to more powerful devices, which can be very far away, in the "cloud" (SZE et al., 2016).

Cloud computing is a very successful model, but it is not a panacea for all smart devices' needs. Sometimes local processing becomes mandatory for many reasons, including:

- Speed: network bandwidth and latency constraints may be unacceptable, especially for real-time systems;
- Availability: a network may not be available where the system is deployed;
- Reliability: the rate of network or remote system failure may be unacceptable;

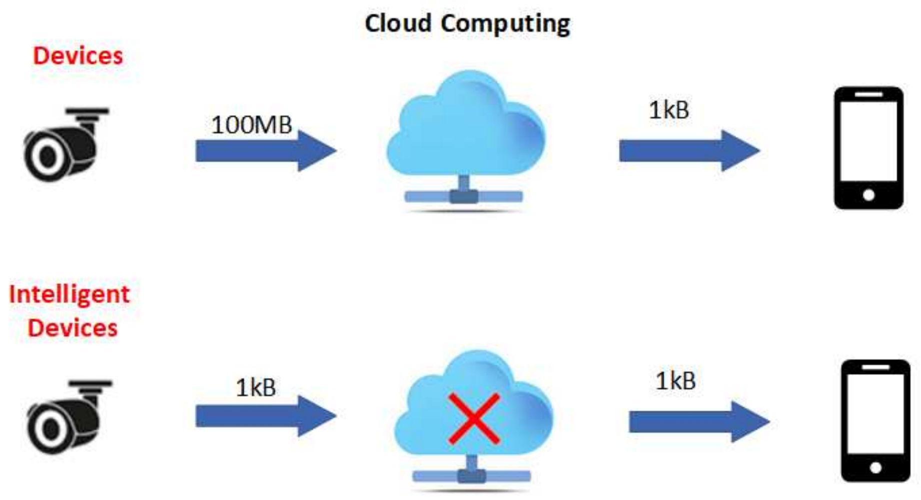


Figure 1.1 – Deep learning in Cloud service and Hardware Accelerator

- Cost: the cost of communication and remote computation may be higher than providing local computational power;
- Privacy and Security: it might be undesirable to transfer sensitive data to remote servers.

In such cases, hardware accelerators may provide an optimal solution for all local computational requirements (Figure 1.2). For desktops and servers, Graphics Processing Units (GPUs) are, by far, the essential hardware accelerators for deep learning. Indeed, the so-called GPGPU (General-Purpose computing on GPUs) has revolutionized high-performance computing, deploying thousands of cores in a cheap, compact form factor.

Most embedded systems, however, cannot support complex GPUs (or *any* GPU, indeed). While GPUs are cost and energy-efficient for servers and desktops, their sheer complexity makes them unfeasible for all but the most complex embedded systems, like high-end cellphones and tablets. On the other hand, these devices - Application Specific Integrated Circuits (ASICs) and Field-Programmable Gate Arrays (FPGAs) - are widely used to perform several tasks in embedded systems, executing efficiently with low power consumption and cost (Figure 1.2).

Therefore, we focused on deploying deep learning for embedded systems with all limitations and restrictions in this work, using continuous authentication as a motivating application.

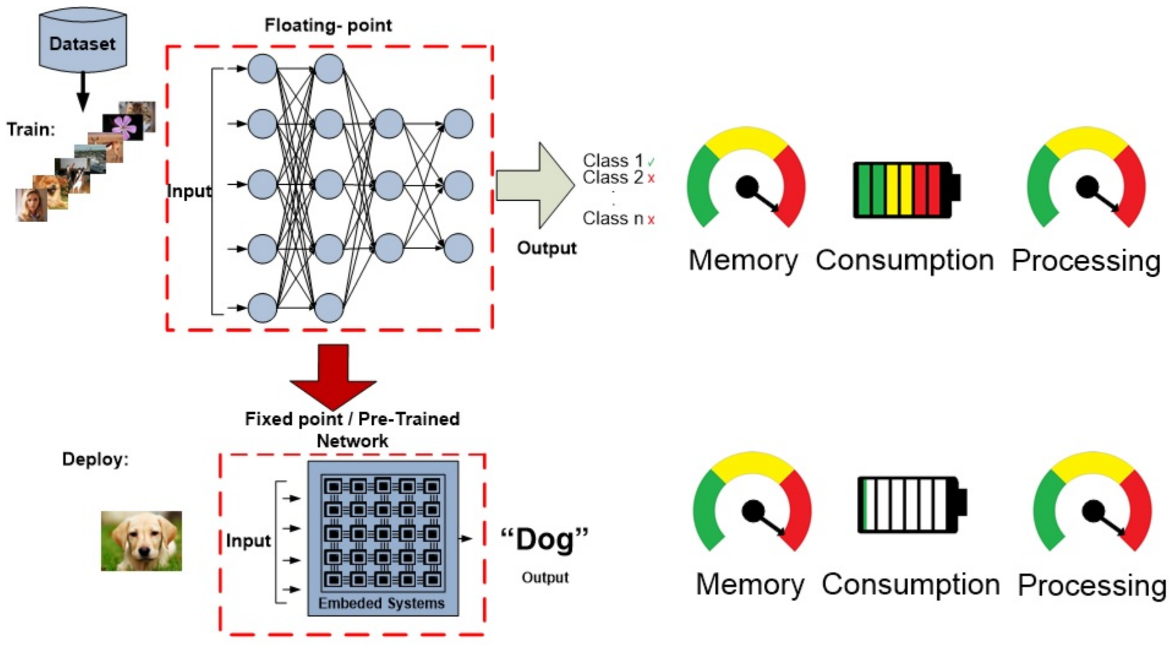


Figure 1.2 – Deep Learning in Hardware accelerator with a power consumption point of view. Figure adapted from (CEVA,).

1.2 Objectives and contributions

Our main goal is to study the implementation of deep learning models on limited hardware, such as that used for embedded systems.

As specific objectives for this project, we have to:

- Proposing a novel hardware architecture for deep learning on hardware.
- Implementing and testing such architecture in an FPGA.
- Use a motivating application (Continuous Authentication) to validate our implementation by comparing its results with a reference software implementation.

1.3 Outline

This remainder of this thesis is organized as follows:

Chapter 2 - Literature Review starts with a brief introduction of deep learning (Section 2.1). We focus on convolutional and recurrent neural networks, examining their foundations. We explore hardware for deep learning, especially in embedded systems, in Section 2.2. In Section 2.3, we briefly present our motivation application of continuous authentication.

Chapter 3 - Proposed Architecture describes *Mini Mind*, the architecture we proposed to accelerate deep learning algorithms in embedded systems.

Chapter 4 - Experiments and Results shows our experiments, in which we contrast two implementations of continuous authentication. The first uses an image RGB based on gesture sequence as a new user authentication method. The second is the Mini Mind design to perform the entire authentication process.

Chapter 5 - Conclusion synthesizes our contributions, with perspectives for future contributions.

2 Literature Review

Deep learning has attracted much attention as a proven method for learning complex, accurate models in large datasets (BENGIO; COURVILLE, 2016). “Deep learning” comprises a significant — and growing — family of models for tasks in essentially all areas of machine learning: supervised, unsupervised, or semi-supervised learning, reinforcement learning, generative models, etc. However, we will focus only on a narrow class of models: convolutional and recurrent neural networks for image classification.

Convolutional neural networks (CNN) are widely used for classification and object recognition. Implementing CNNs is highly challenging because the convolutional layers are very greedy in terms of computation and memory bandwidth (ZHANG et al., 2015). Recurrent neural networks (RNN), which are often used for sequences of data or time series, present, thus, an exciting alternative for implementing simpler networks.

We divided this chapter into four sections: In the first, we survey the history of deep learning history until the current state-of-the-art of deep learning, focusing on CNNs and RNNs. In the second, we focus on the core of our work, deep learning on embedded systems, especially FPGAs as accelerators. Next, we survey some cutting-edge developments of deep learning in embedded hardware, which appeared in the last year, after the development of our main results was well advanced. We focus on hardware development in this work because it is a much more onerous endeavor than software development. Thus, those developments could not be incorporated into our experiments (Such as deep learning model optimization), but we comment on them to give the reader a contemporary perspective of the area. Finally, we briefly survey the area of our case study application, user authentication in low power devices, with the most relevant works in the area.

2.1 Deep Learning History

Artificial intelligence is a thriving subject with a long and broad history, which includes the application of patterns recognition to many fields. In 1943, (MCCULLOCH; PITTS, 1943) introduced the first artificial model of the neurons and synapses of the human brain. Based on this work, (HEBB, 1949) developed *Hebb’s rule* in his work *The Organization of Behavior*, considering the first training method for Artificial Neural Networks (ANN).

In 1958, (ROSENBLATT, 1958) introduced a new way to illustrate the artificial neural network (Backbone of AI) called *Perceptron* as shown in Figure 2.1a. The perceptron was a binary classifier that used a linear decision boundary to predict an output. Perceptron's design was the first generation of neural networks. Based on that, (WIDROW; HOFF, 1962) and (WIDROW, 1964) developed *Adaline* and *Madaline* (Multiple Adalines) networks for real applications. Those networks have better converge properties than the perceptron due to stochastic gradient descent (SGD) in the learning phase. SGD was introduced by (ROBBINS; MONRO, 1951), and modified to its more recognized form in Machine Learning by (KIEFER; WOLFOWITZ et al., 1952).

In 1969, (MINSKY; PAPERT, 1969) established that Rosenblatt's perceptron — and almost all ANN models of the time — could not learn straightforward tasks (e.g., exclusive-OR), leaving a loss of interest in neural networks for many years. Such long periods of loss of interest would become cyclical in the area and later denominated the "winters of AI".

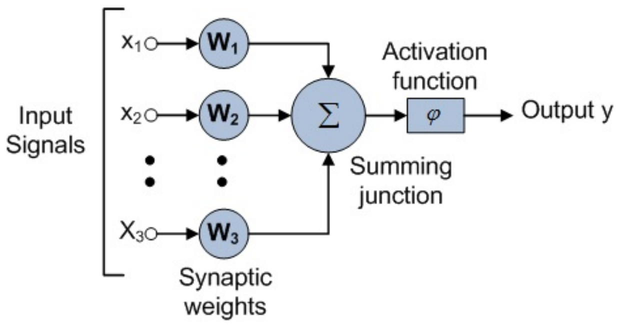
The first winter began to thaw with the introduction of a new learning algorithm called *backpropagation* (WERBOS, 1974). Backpropagation is an efficient method for using derivatives to propagate the error signal — which is computed only on the last layer — to earlier layers of a neural network, allowing to update the weights on every layer (WERBOS., 1990). That work spurred renewed attention on ANN, showing how large neural networks could be efficiently trained and solving previous problems introduced by Minsky's work.

In 1975, (FUKUSHIMA, 1975) introduced a new feature extractor in neural layers to recognize patterns called *Cognitron* but it had difficulty in achieving invariance to translation. However, five years later, based on Cognitron's architecture, (FUKUSHIMA., 1980) introduced *Neocognitron*, solving previous problems in Cognitron design which can recognize the shape of the pattern independently of it is size and position. Fukushima's works are considered the firsts algorithms of Convolutional Neural Networks.

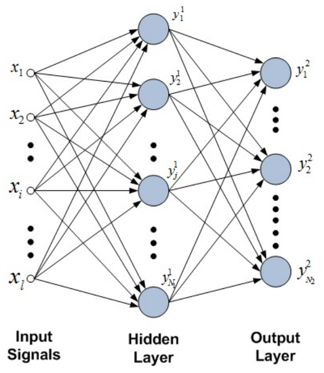
In 1982, (HOPFIELD, 1982) introduced another architecture of neural network called *Hopfield Network*, inspired by the biological interaction between neurons but implemented with mathematical operations. That feed-forward neural net was called *energy based*, and its properties derive from a global energy function. This network is composed of binary threshold units with a recurrent connection between them, and it can store memories as distributed patterns of activity.

In 1985, (RUMELHART; HINTON; WILLIAMS, 1985) introduced Multi-Layer Perceptrons (MLP) (Figure 2.1b). It was a general model to represent complex neural networks with many internal layers (called *hidden layers* in literature). Three years later, (RUMELHART et al., 1988) introduced a parallel distributed processing (PDP) idea. It was a framework for representing and understanding the computing potential of neural networks. With those ideas, we reached the second generation of neural networks.

Many studies on pattern recognition and classification in many other fields



(a) Perceptron by Rosenblatt. Figure reproduced from (OMONDI; RAJAPAKSE, 2006).



(b) Multi-layer Perceptron. Figure reproduced from (OMONDI; RAJAPAKSE, 2006).

Figure 2.1 – Neural Networks

encouraged the third generation of neural networks. For example, in natural language processing, (JORDAN, 1986) introduced recurrent neural networks (RNN) by creating the idea of *recurrence*. This neural network understands time sequences with lexical categories across different classes of items. Previous networks followed the *feed-forward* principle: the input is fed to the first layer of the network, whose output is fed to the second layer, whose output is fed to the third layer, etc., until the final output, without feedback. Recurrence allowed a feedback, creating paths in which the output of a layer could be used as an input to a previous layer. In a recurrent network, there is a *time dependence* on the output since the output of a layer may chance according to the feedback it has received for a last layer on the previous timestep. The first recurrent networks were called Jordan's Networks and were soon followed by Elman's networks (ELMAN, 1990). The main difference between Jordan's and Elman's networks was distributing the information in the recurrent paths. In Elman's networks, the hidden layer feeds retaining memory of past inputs, but in the case of Jordan's networks, they store the output layer into the state layer.

Before 1995, neural networks had several problems and limitations when data were correlated (as is the case for images). Therefore, (LECUN; BENGIO, 1995) developed another type of neural network called *convolutional neural networks* as they are currently known. Combining local fields, sharing weights, and making spatial subsampling, those networks learn some representations (Edges, lines, or corners) using filters, sharing that feed information to other learned filters until, later in the network, conventional (non-convolutional) layers, make the final decision. The convolutional model would, two decades later, become state of the art in almost all image recognition tasks.

RNNs used Stochastic Gradient Descent (SGD) to learn sequence or time-series prediction. However, (BENGIO; SIMARD; FRASCONI, 1994) showed that it could be challenging to train an RNN for long-term sequences because the gradient of the loss function decays exponentially with time (Called the vanishing gradient problem). Therefore, in 1997, (HOCHREITER; SCHMIDHUBER, 1997) introduced a novel efficient algorithm called *Long Short-Term Memory* (LSTM). This new architecture could train very long time series with the use of logic cells (variation to the traditional RNN architecture), retaining more information through time.

In the mid-2000s, the first neural networks to tackle large-scale complex problems appeared. Those models, appropriately, required increasing amounts of data and computation to converge. (HINTON; OSINDERO; TEH, 2006) developed Restricted Boltzmann Machine (RBM), the first real "deep" network to achieve success. RBMs used a faster algorithm than backpropagation, which allowed it to require less data and computation than a traditional MLP architecture. Those models are also called the third generation of neural networks.

In 2012, (HINTON et al., 2012) introduced *dropout* regularization to avoid overfitting in neural networks. It consisted of dropping out some neurons, preventing complex adaptations, and learning more representative data during the training set, showing significant improvements when used in feed-forward architectures, e.g., Convolutional Neural Networks ((KRIZHEVSKY; SUTSKEVER; HINTON, 2012)). In the same year, (KRIZHEVSKY; SUTSKEVER; HINTON, 2012) won the ImageNet Challenge — a competition involving more than a million images divided into a thousand classes — with Alex-net architecture. They used GPUs to accelerate the computation of CNNs, and Hinton's dropout to prevent overfitting. That victory was a watershed for deep-learning models in computer vision: soon after, deep learning became the main focus of most research groups worldwide.

From a computational point of view (Table 2.1), Alex-net has 61 Million parameters (249MB of memory) and performs 1.5 billion of 32-bit floating-point operations to classify one single image (RASTEGARI et al., 2016), spending 90% of those operations on the convolutional layers. That makes its implementation in embedded hardware challenging. Moreover, with its dozen or so layers, Alex-net is very shallow for today's standards.

Indeed, models have progressed steadily to increasing "deeper" learning (SZE et al., 2017) with an ever growing number of hidden layers, reaching up to hundreds of layers (SZEGEDY et al., 2015; HE et al., 2016) Correspondingly, the number of operations, parameters (weights) — and accuracy — of models has also been steadily increasing (Table 2.2).

Deep learning algorithms require large amounts of data for training, which is not always feasible (In most cases). Therefore, *Transfer Learning* is often employed, which consists of pre-training a model in a task for which data is readily available, and then refining it for another where data is scarce (PAN; YANG, 2010). By recycling knowledge from one task to another in learned model weights, transfer learning made it possible to apply deep learning in virtually all image classification tasks.

Table 2.1 – Number of operations of every single layer in Alex-net architecture. Table adapted from (SUDA et al., 2016)

Layer	**Features**	**Operations**
Input image	224	
Convolution-1/ReLU-1	96	**211,120,800**
Normalization-1	96	3,194,400
Pooling-1	96	629,856
Convolution-2/ReLU-2	256	**448,084,224**
Normalization-2	256	2,052,864
Pooling-2	256	389,376
Convolution-3/ReLU-3	384	**299,105,664**
Convolution-4/ReLU-4	384	**224,345,472**
Convolution-5/ReLU-5	384	**149,563,648**
Pooling-5	256	82,944
Fully connected-6/ReLU-6	4096	75,501,568
Fully connected-7/ReLU-7	4096	33,558,528
Fully connected-8	1000	8,192,000
Total Operations		**1,455,821,344**

Table 2.2 – Summary of popular deep neural networks. Table adapted from (SZE et al., 2017)

Metrics	**Alexnet**	**Overfeat – Fast**	**Vgg – 16**	**Googlenet v1**	**Resnet 50**
Top-5 error	16.4	14.2	7.4	6.7	5.3
Input Size	227 x 227	231 x 231	224 x 224	224 x 224	224 x 224
Filter sizes	3,5,11	3,5,11	3	1,3,5,7	1,3,7
# of Conv. Layers	5	5	13	57	53
# of FC. Layers	3	3	3	1	1
Total Weigths	61M	146M	138M	7M	25.5M
Total Operations	1.45G	5.6G	31G	2.86G	7.8G
Year of Publication	2012	2014	2014	2015	2015

2.2 Deep learning in Embedded Systems

In the history of neural networks in hardware, the primary focus has been to obtain a design with high speeds at low costs. That implies exploiting the maximum computational power of as few as possible hardware electronic elements. An additional design goal is fault tolerance, where the system degrades “gracefully” when facing adverse situations but might require some level of redundancy (Parallel and pipelined architecture).

According to the survey of (MISRA; SAHA, 2010), the last two decades witnessed many approaches to neural networks in hardware: digital, analogic, hybrid, and, finally, FPGA-based. Of all alternatives, FPGAs proved an excellent choice for quick implementation due to their low cost and reconfigurable flexibility. However, implementing large models with thousands of neurons is still a research challenge.

(SHAWAHNA; SAIT; EL-MALEH, 2019) summarizes the most relevant neural network implementations on FPGAs, comparing techniques used to optimize resource utilization such as loop tiling, loop unrolling, and data reuse (memory or essential elements), and operation pipelining.

In the remainder of this section, we will use the most relevant works in literature (MISRA; SAHA, 2010; SHAWAHNA; SAIT; EL-MALEH, 2019; ABDELOUAHAB et al., 2018) to develop our state-of-the-art of deep learning algorithms implementation on FPGA.

The first implementation of neural networks in FPGA was called *Ganglion* (COX; BLANZ, 1992). Ganglion's architecture has 12 units in its input layer, 14 units in its hidden layer, and four units in its output layer. That architecture is based on a feedforward neural network, which is easier to implement in hardware than the Hopfield network due to fewer interconnections, fewer units in the hidden layer, and an activation function that can be determined immediately at the end of each unit.

$$x_j^l = \sum_{n=i} A_{ij}^l * S_{ij}^{l-1} + B_j^l \tag{2.1}$$

Ganglion architectural contribution was the multiplication/accumulation between inputs and weights — Units in a feedforward neural network shown in Equation (2.1). (COX; BLANZ, 1992) proposed to break a complex multiplication (8×8 bits) into two smaller multiplications (8×4 bits) and one addition, as shown in Figure 2.2. In this process, Ganglion uses two bytes as input S and A, and also U and L to represent nybbles. The constants values 128 and 2048 are considered to guarantee the product in this process. With that new approach, the Ganglion can compute 224 fixed-point multiplications per clock, requiring fewer logic circuits to implement in hardware. Ganglion's architecture opened a new era for neural networks in embedded hardware because it shows another way to represent a complex multiplication process into a simple one with fewer elements in hardware.

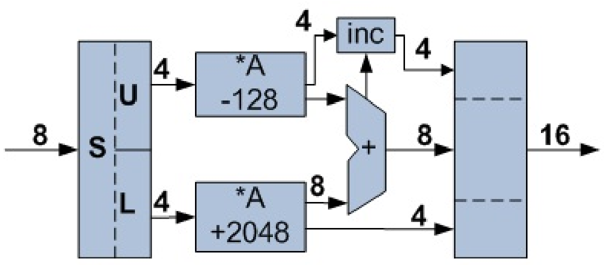


Figure 2.2 – Implementation of Cox multiplication, where S and A are input bytes, and U and L represent nybbles. In this architecture, (COX; BLANZ, 1992) break a complex multiplication of 8 bits into one simple multiplication of nybbles and one addition. Figure reproduced from (COX; BLANZ, 1992)

.

2.2.1 Convolutional Neural Networks in Embedded Systems

The first implementation of a CNN in FPGA was the *Virtual Image Processor* (VIP) (CLOUTIER et al., 1996). VIP had to control the numerical precision among the convolutional operations and still had to use 5 FPGAs to gather - at that time - enough elements to implement the CNN. In each multiplication unit, an FPGA uses an embedded multiplier and also uses logic gates to do interconnections among other FPGAs. Figure 2.3 shows the VIP architecture, which is composed of three components: first, the processing matrix, which has the processing elements (PEs). Second, the SIMD (Single Instruction Multiple Data) controls executing instructions among the processing element (PEs) to perform matrix and vector operations. And finally, the input/output controller. The matrix of PEs was implemented with a low precision provided to have an excellent performance in hardware than the full precision in multipliers. In applications such as image processing and pattern recognition, where RAM is a critical factor, this new approach of VIP implementation uses only 1.5 MB of static memory RAM.

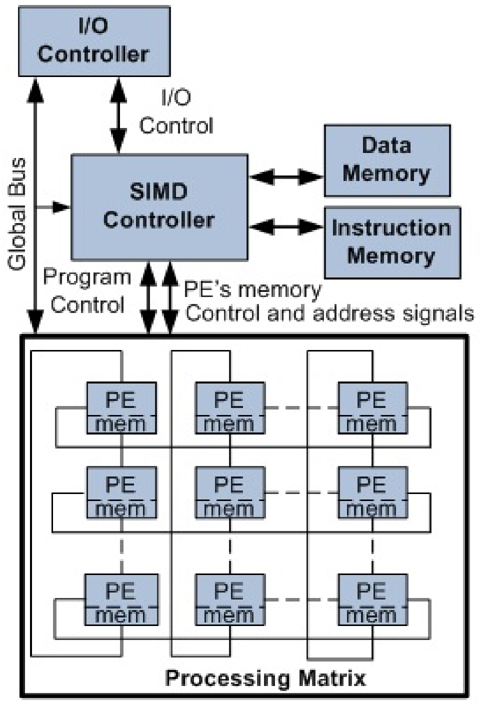


Figure 2.3 – Implementation of Cloutier is composed of: first, the processing matrix, which has the processing elements (PEs), SIMD control to execute instructions PEs and the input/output controller. Figure adapted from (CLOUTIER et al., 1996)

In a few years, (FARABET et al., 2009) proposed the use of small kernels filters (11 × 11 for a single kernel, 7 × 7 for two simultaneous kernels) and a pre-trained neural network for real-time applications. The efficiency of that new architecture came from performing multiplication and accumulation simultaneously at each clock cycle. They employed 16-bit fixed-point arithmetic operations and 8 bits for the states. They were able to implement LeNet-5 (FARABET et al., 2009), a famous early CNN architecture known for its high performance on the MNIST handwritten digit recognition challenge (LECUN et al., 2015). Lenet-5 has three convolutional layers, two polling layers, and one fully connected layer. This work marks a significant turning point in using FPGAs as accelerators for artificial neural networks embedded vision applications.

For the following years, embedded hardware gained attention, employing General Purpose Processors (GPP), ASICs, GPUs, and FPGAs. Each of those platforms has a particular characteristic, depending on the type of application. For example, GPPs are highly flexible and allow easy and fast implementation of algorithms. ASICs provide high performance, low power consumption, and small form-factors but are inflexible and have substantial development costs. GPUs allow massively parallel implementation of algorithms but have high per-unit costs and significant power consumption needs. In contrast with these platforms, FPGAs have challenges but offer an exciting trade-off between performance, power consumption, and programmability.

In deep learning, an FPGA design focuses on adapting algorithms for a fixed computational structure, allowing more freedom to explore low-level optimizations than in software. Optimization allows reusing resources to minimize the logical elements required to deploy the hardware. Regarding disadvantages, an FPGA design requires many complex low-level hardware control operations, which are often problematic and extremely difficult using hardware description languages like VHDL or Verilog. However, in a complex embedded system, the hardware provided by FPGA in terms of throughput, hardware elements, and power consumption has made them the standard tool for real-time processing applications.

In the past decade, deep learning models have steadily grown in size and complexity, thus requiring more memory blocks and logic elements to be implemented in hardware. The primary focus of their implementation in FPGAs was the data flow during computations to extract the highest possible throughput from the limited number of components. In this context, (GOKHALE et al., 2014) presented another FPGA-based accelerator called nn-X, containing a register for weights in MAC units in direct communication to a global buffer with all the network weights. That configuration minimizes energy consumption for reading the weights and maximizes the use of registers. The design of nn-X comprised a coprocessor, a host processor, an external memory, 16 bits fixed-point precision multipliers (with 8 bits for the integer and 8 bits for the fraction part), and piecewise linear approximation for the nonlinear functions. A significant advance of this work was the use of the coprocessor to implement large convolutional units in parallel, using pipelines (Figure 2.4), allowing it to process convolutional neural networks for vision tasks (Images or videos) in real-time.

For implementing neural networks, the numerical precision of multipliers is

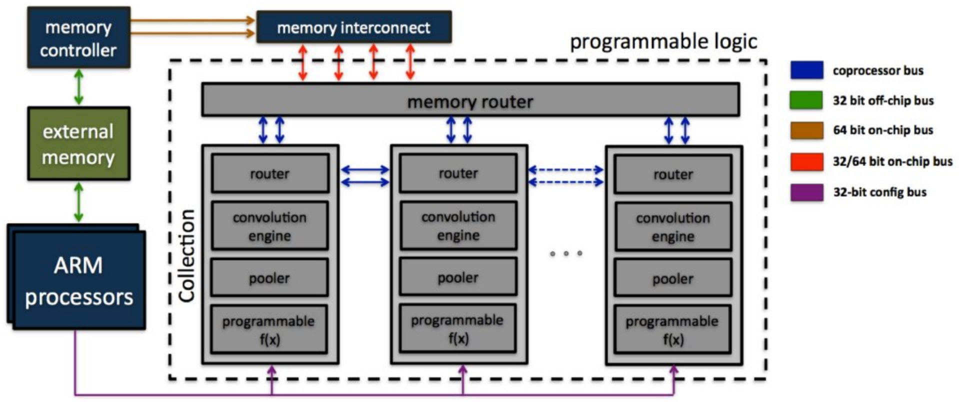


Figure 2.4 – Implementation of Gokhale et al. Figure adapted from (GOKHALE et al., 2014)

a critical factor since fixed-point arithmetics will typically be much faster and more resource-consuming than floating-point. (COURBARIAUX; BENGIO; DAVID, 2014) contrasted model accuracy using three precision formats, as shown in Figure 2.5. The first precision format, floating-point, is more often used for real values because it uses a sign, an exponent, and a mantissa to represent any number. Second, fixed-point uses signed mantissa, and a global scaling factor is shared between all fixed-point variables. Moreover, finally, dynamic fixed point, based on point format in which there are several scaling factors instead of a single global one. They found that with 10 bits of fixed-point precision, it was possible to train a Maxout network, a feedforward network with max-pooling activation unit (GOODFELLOW et al., 2013). They also contrasted single-precision (32 bits) vs. half-precision (16 bits) floating-point, finding that one can use the latter without much impact in the training phase.

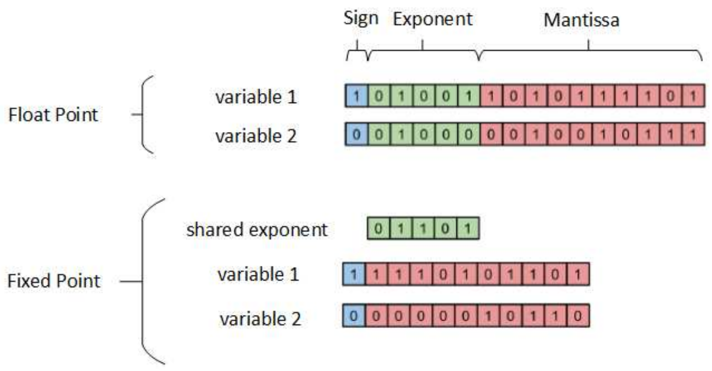


Figure 2.5 – Description of implementation floating and fixed point, using two signs variables showing sign, exponent and mantissa distributions. Figure adapted from (COURBARIAUX; BENGIO; DAVID, 2014)

(GUPTA et al., 2015) also studied the effect of limiting precision on neural network training, concluding that the error in a training phase using only 16 bits fixed point was equal to having 32 bits floating point operation using stochastic rounding. More recently, (CARVALHO et al., 2016) proposed a new method for compressing a deep learning architecture by reducing the feature vectors and quantizing the remaining elements, obtaining a compression rate of 98.4%. Those works suggest a large allowance in the trade-off between numerical precision and model accuracy, both during the training and inference phase, to better implement the networks in hardware.

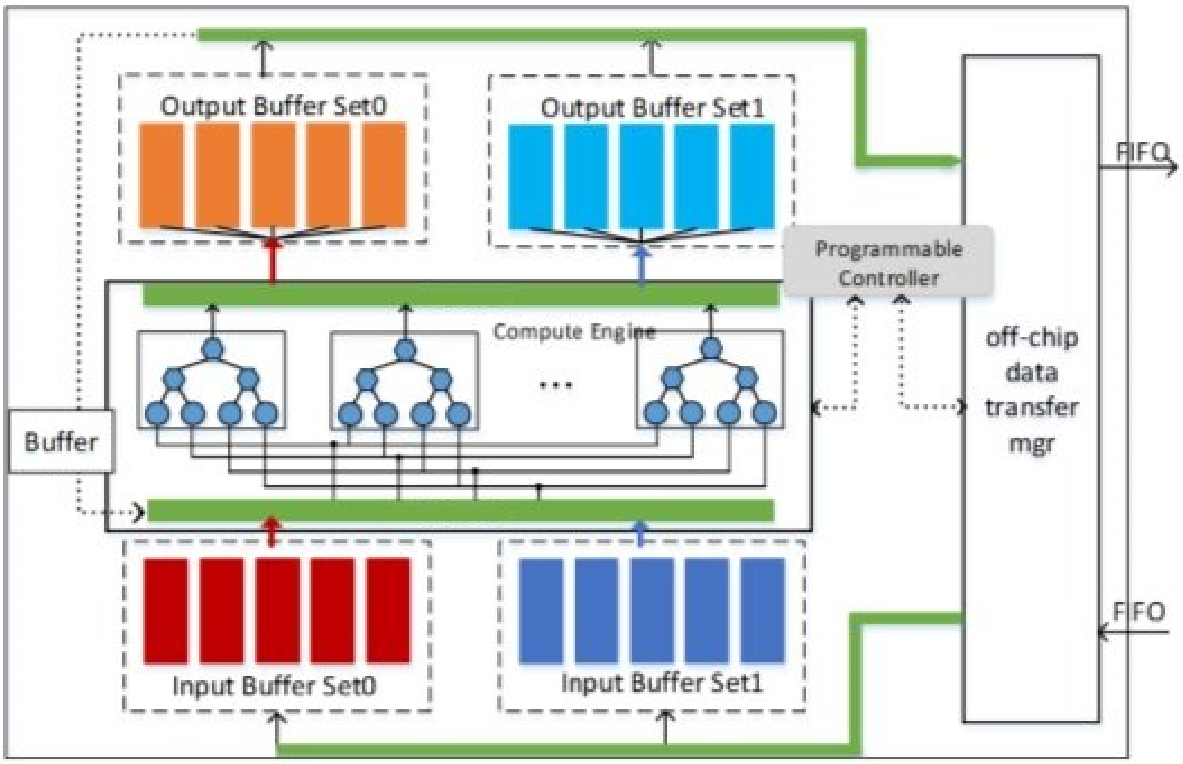


Figure 2.6 – Implementation of Zhang et al. Figure adapted from (ZHANG et al., 2015)

(ZHANG et al., 2015) proposed a novel hardware accelerator using the Roofline model (a metric that relates memory traffic and peak performance) to choose the design with the best throughput. Figure 2.6 shows the basics components in that design, such as processing elements (PEs composed of adders and multipliers), on-chip buffer, and external memory. The significant advance in that design was implementing the PEs units in a pipeline and extensively buffering memory (Input, weights, and output) among computing processes, thus minimizing external memory accesses. (ZHANG et al., 2015) compared their results with several others with similar designs using off-chip memory access, buffered input and output, and parallel processing (Table 2.3).

In the modern design of hardware accelerators, each layer of a network is seen as a collection of multiplications and accumulations. Thus, the parallelism of deep networks is limited by the number of multipliers available on the FPGA (embedded multipliers or MAC units). To deal with that issue, many authors proposed to reuse hardware to implement deep learning architectures, such as that aimed at ImageNet (Table 2.1). (RAHMAN; LEE; CHOI, 2016) implemented a 3D neuron array design with the use of a new technique called Input-Recycling Convolutional Array of Neurons (ICAN). That new architecture can maximize the use of multipliers, applying one layer at a time, which is very efficient regarding resources. In addition, in the same year, (DUNDAR et al., 2016) proposed to optimize resources in hardware with the use of data concatenation without causing any delay. The use of data concatenation increased the maximum speed-up among feature maps used for object classification and object detection in real-time.

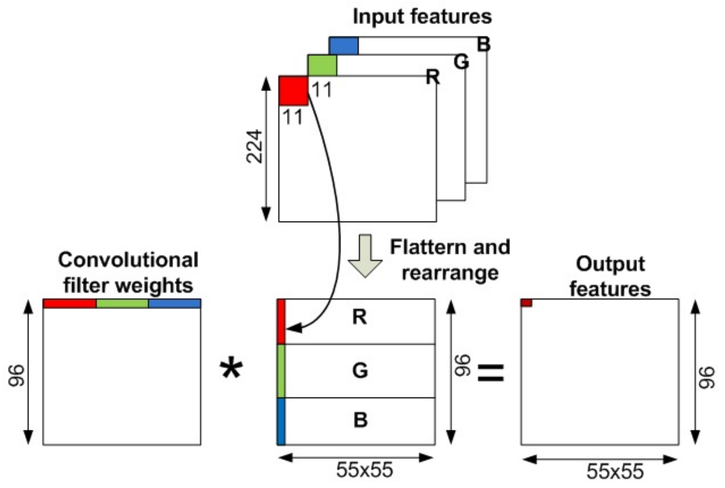


Figure 2.7 – Implementation of Suda. Figure adapted from (SUDA et al., 2016)

In order to find another way to multiply weights and inputs, (SUDA et al., 2016) proposed the idea of flattening and rearranging input features, thus optimizing memory usage. Figure 2.7 shows Suda implementation where the multiplication between the matrix of convolutional weights and the rearranged matrix of inputs are loaded in local memory, reducing the interaction with external memory and increasing performance. (LI et al., 2016) employed the idea of *network in network* (LIN; CHEN; YAN, 2013), implementing a recurrent convolutional neural network (RCNN), another way to implement CNN in hardware, and applied a global summation instead of a fully connected layer for classification. This implementation approach uses a pipeline form architecture, which requires fewer operations, solves the memory access bottleneck problem, and dramatically enhances parallelism.

(COURBARIAUX; BENGIO; DAVID, 2015) adapted convolutional networks to bypass multiplications and minimize memory usage by employing binary values instead of continuous values/weights. Based on that work, (RASTEGARI et al., 2016) presented Xnor-Net, an architecture for the ImageNet task with binary weights and inputs. Xnor-net is based on shift operations and dot products between two binary values (+1,−1). The latter can be implemented by using the *and* operator and counting the active bits — two operations that are usually very fast in current hardware. Binary operators are the most promising way to approximate convolutional layers in limited hardware.

(HAN; MAO; DALLY, 2015) proposed Deep Compression, a method to compress

Table 2.3 – Performance of most important papers of Deep Learning in FPGAs.

	Precision	Type	GOPs [1]	Power[w]
(LI et al., 2016)	Fixed	Object Classification	409.62	1
(DUNDAR et al., 2016)	Fixed	Object Classification	247.00	4
(SUDA et al., 2016)	Fixed	AlexNet	136.50	19
		VGG	117.80	
(RAHMAN; LEE; CHOI, 2016)	Float	AlexNet – CNN	80.78	
(ZHANG et al., 2015)	Float	AlexNet – CNN	61.62	18
(PEEMEN et al., 2013)	Fixed	Sing Recognition	17.00	15
(CHAKRADHAR et al., 2010)	Fixed	CNN – Classification	16.00	14
(CADAMBI et al., 2010)	Fixed	CNN – Classification	7.00	14
(FARABET et al., 2009)	Fixed	LeNet- 5	5.25	15
(SANKARADAS et al., 2009)	Fixed	Face Detection	6.74	11

neural network models up to 59x without losing much accuracy. That process combines a pipeline, a pruning phase, weight quantization, parameter sharing, and Huffman coding. With that method, (HAN et al., 2016) proposed an efficient inference engine (EIE) accelerator for CNNs and image recognition. That new type of accelerator can perform modern networks such as AlexNet or VGG-16 because of weights compression (Prunning and weight sharing). This architecture uses sparse matrix-vector multiplication, reporting 600 mW power and achieving 102 GOPS on a compressed network.

More recently yet, (WANG et al., 2017) proposed the DLAU design, which is a scalable accelerator for large-scale deep learning networks using an FPGA as a platform. The DLAU accelerator uses three units in the pipeline, dividing a considerable input into small subsets with FIFO buffers to increase the throughput.

2.2.2 Recurrent Neural Networks in Embedded Systems

RNN architectures are often the most appropriate for serial data (e.g., speech), as they naturally provide sequences both as input and output. As explained in Section 2.1, RNNs differ from feed-forward architectures, by providing *recurrent links* to previous layers.

One of the most used RNN architectures is LSTMs, which, compared with traditional RNNs, contain temporal memory blocks in their recurrent hidden layer. Those memory "cells" contain self-connections that store a temporary state (output at time t). (GREFF et al., 2015) proposed another architecture evolution, removing the peephole connection of LSTMs, without suffering any loss of accuracy. Based on that, (CHANG; CULURCIELLO, 2017) implemented a Vanilla LSTM architecture in hardware, for learning character text sequences, using 16-bit fixed-point numerical precision (Figure 2.8). Their adapted LSTM architecture has three gates (input, forget, and output), block input, a single cell (constant error carousel), a sigmoid output activation function for input, output, forget gate, and a hyperbolic tangent activation function for the candidate memory. The input gate controls the flow of input activations into the memory cell, the output gate controls the output flow into the rest of the network, and the forget gate selects which value is relevant to store in the cell's memory. That new architecture follows these equations:

Input gate:

$$i_t = \sigma(W_{xi} * x_t + W_{hi} * h_{t-1} + b_i) \tag{2.2}$$

Forget gate:

$$f_t = \sigma(W_{xf} * x_t + W_{hf} * h_{t-1} + b_f) \tag{2.3}$$

Output gate:

$$o_t = \sigma(W_{xo} * x_t + W_{ho} * h_{t-1} + b_o) \tag{2.4}$$

Candidate memory:

$$\tilde{c}_t = tanh(W_{xc} * x_t + W_{hc} * h_{t-1} + b_c) \tag{2.5}$$

Memory cell:

$$c_t = f_t \odot c_{t-1} + i_t \odot \tilde{c}_t \tag{2.6}$$

Output vector, the final result of LSTM unit:

$$h_t = o_t \odot c_t \tag{2.7}$$

Where $W_{x\_}$, $W_{h\_}$ and $b\_$ are the trainable network parameters — input weights, previous output weights, and biases respectively — of each ($i$nput, $f$orget, $o$utput, $\tilde{c}$andidate, $c$ell, $h$output) gate; x_{t-1} is the input vector; h_{t-1} is the output vector of the previous timestep; σ is the logistic sigmoid function; and $\odot$ is the elementwise vector product, also known as Hadamard product. Figure 2.8 shows a new implementation without peepholes connections, which requires fewer elements in hardware, opening the possibilities for many applications.

(HAN et al., 2017) proposed another hardware accelerator called an efficient speech recognition engine (ESE) for the LSTM network with a bit of variation in the base process. For example, during pruning, they forced each row to have the same amount of weights to implement hardware load balancing. They also proposed to compress the LSTM model by *x20* without sacrificing the prediction accuracy; working directly on the compressed model enables ESE to have a significant advantage of parallelism.

The work proposed in this thesis, Mini Mind, uses hardware acceleration for a continuous authentication process (Section 3. Our solution is based on implementing an LSTM with a pipeline, using gesture information (Temporary sequence generated by a user, Section 2.3) as input data to authenticate a user.

In terms of hardware-description languages (HDL) for the implementation of neural networks, open computing language (OpenCL) has gained attention for CPUs, GPUs,

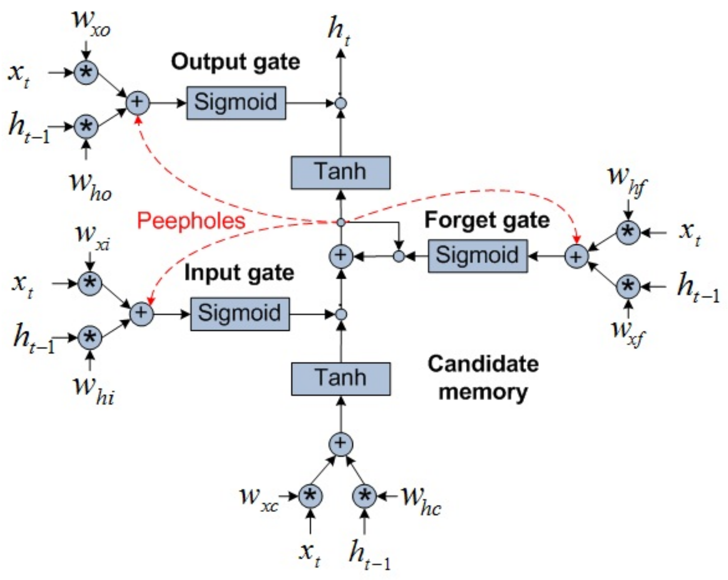


Figure 2.8 – Vanilla LSTM architecture. Reproduced from (CHANG; CULURCIELLO, 2017)

DSPs, and most recently, FPGAs, allowing to implement complex parallel architectures in a short time of design. OpenCL is a high-level language based on C (used for software), which can then be processed by synthesis tools to be used as an HDL. More traditional HDL choices are VHDL and Verilog, but for complex designs, they require elaborate simulations (Design, test, power, and resource estimation), and expert knowledge to allow implementation in an FPGA target. In contrast, OpenCL allows design and verification to be integrated. Thus, design time using OpenCL is up to six times faster than using traditional HDLs. Still, when efficiency is at stake, traditional HDL gets the upper hand regarding the use of logic resources with 59% to 70% less logic, while they are maintaining similar timing constraints (HILL et al., 2015). Because of those efficiency advantages, we decided to make our implementations using VHDL in this work.

2.2.3 Latest Developments and and Future Directions

In this section, we will survey the most recent developments in the literature of CNNs and RNNs on hardware, covering the past year until the publication of this thesis. We cover those works separately due to the nature of the development of this work in the hardware. Contrary to experiments performed in software, development in hardware is much slower and cannot hope to be up-to-date with literature that is months old. Thus, although presenting an exciting new frontier, the works covered in this section could not be incorporated into our design. We hope they will inform future versions of our work.

(ABDELOUAHAB et al., 2018) mentions that the newest implementations in hardware focus on the inference phase in order to bypass the complexity of training. Indeed, computing the forward pass in a neural network is cheaper than computing the gradients required for backpropagation.

(WANG et al., 2019) summarizes 18 deep learning frameworks and libraries. Those frameworks proved to be efficient in timely execution and precision and flexible in numerical precision, allowing the choice between floating-point and fixed-precision and between 32 and 16 bits precision. For research purposes (Based on GitHub community statistics) are: Pylearn2, Tensorflow, Pytorch, Caffe, Keras, and Tensorflow are the most commonly used due to flexibility, debugging facilities, and ease of use. Deeplearning4j and Matlab are more stable and scalable than the other frameworks for industry use. With this type of advances, in the following years, not only in the mobile environment, those frameworks will have a significant impact on platforms such as FPGA and ASIC.

In terms of quantization, Angel-Eye (GUO et al., 2018) is a model which uses conventional floating-point numbers, then in each layer, collects the statistics of feature maps, especially the histogram of the parameters (weights), to calculate the best radix position to represent those values in fixed-point numbers. They fix that radix-position for all layers for adders and multipliers, thus reducing precision to 8 bits with negligible accuracy loss in VGG-based architectures, with 16x better energy efficiency.

(JIANG et al., 2019) proposed a new model of hardware implementation called FNAS (hardware-aware NAS framework), which emphasizes latency reduction, and allows training the network in the FPGA as an option. The Neural Architecture Search (NAS) frameworks proposed a new way to create architectures of neural networks. NAS uses reinforcement learning and an evolutionary algorithm to search for the best possible solution, which means it can find the optimal architecture of a neural network given a dataset. However, most of them take a long time to find the optimal architecture due to the vast search space to train and evaluate each candidate. The authors report that in a dataset such as ImageNet, the FNAS architecture achieves 11.13× of speedup for the search process, a 7.81× of latency, and less than 1% of accuracy loss.

Most authors propose weight pruning techniques to reduce computational costs in previous works. Unfortunately, irregular computation and memory access in sparse LSTM memory limit the realizable parallelism. To address that issue, some researchers propose block-based sparsity patterns to increase the regularity of sparse weight matrices, but those approaches suffer from deteriorated prediction accuracy. That is why (CAO et al., 2019) proposed Bank-Balanced Sparsity (BBS), a new sparsity pattern that can maintain model accuracy during the FPGA implementation. That new architecture contains a sparse matrix-vector multiplication unit (SpMxV Unit), an element-wise vector operation unit (EWOP Unit), a direct memory access module (DMA) for load/store operations on-chip memory, and a central controller. BBS design improves by 3.7× on energy efficiency and a reduction by 34.4× on latency with negligible loss of model accuracy in the best scenario.

2.3 Selected Application: Continuous Authentication

User authentication is critical to enforcing security in computer systems. In mobile devices, traditional methods of security for user authentication are passwords, personal identification numbers (PINs), or secret questions. However, those techniques are susceptible to multiple types of attacks, as users are notoriously bad at choosing good passwords and keeping them secret.

In recent years, alternatives to passwords for authentication have appeared based on users' biometry — intrinsic biological characteristics. The most popular biometric systems recognize users' fingerprints, faces, voices, irises, and retinas.

In addition to single-point user authentication, continuous authentication recently emerged as an alternative to make mobile devices more secure. It uses biometric techniques to continuously evaluate user authentication in real-time after the first access (PATEL et al., 2016). It uses sensors in a mobile environment such as a gyroscope, touch screen, accelerometer, orientation sensor, and pressure to identify the user. Gestures and gait are common biometric markers studied for continuous authentication.

Table 2.4 – User Gesture occurred on the screen's horizontal axis (Figure 2.9 and Eq. 2.8). Pressure and area values are constant between the same user but different among other users

Action	X	Y	Time	Pressure	Area	Orientation
0	213	285	0	0.41	0.04	1
2	209	287	10	0.46	0.04	1
2	202	291	21	0.52	0.04	1
2	194	294	32	0.56	0.04	1
2	186	297	48	0.56	0.04	1
2	139	309	110	0.65	0.04	1
2	129	312	112	0.65	0.04	1
2	111	318	135	0.65	0.04	1
2	96	326	160	0.65	0.04	1
2	84	332	184	0.65	0.04	1
2	75	337	210	0.38	0.04	1
2	69	341	234	0.18	0.04	1
1	69	341	236	0.18	0.09	1

Gesture recognition is one of the most commonly used techniques for continuous authentication. It is based on the hand and finger interaction with a touchscreen. A single gesture considers how the user makes a finger swipe on a touchscreen while performing an essential operation (Figure 2.9 and Eq. 2.8). Gesture recognition must work in low-power devices (mobiles), using touchscreen interactions as input data, converting the gesture sequence information into feature vectors used for authentication purposes. Each gesture can provide a sequence of data such as:

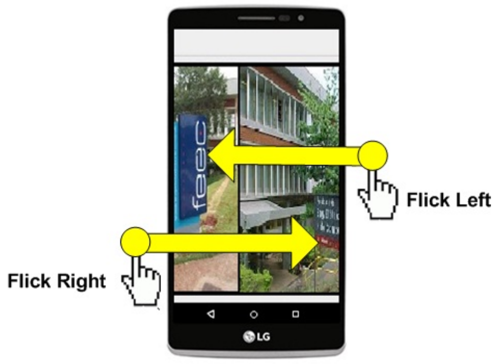


Figure 2.9 – Gesture of gesture data

$$S_n = (T_n, X_n, Y_n, P_n, A_n, O_r),\ n = \{1, 2, 3..n\} \tag{2.8}$$

- Timestamp: T_n
- Position: X_n,Y_n
- Pressure (calculated): P_n
- Area of the finger on the screen: A_n
- Orientation of the cellphone: Or

Gesture recognition works because people tend to be consistent in interacting with the device. For example, some users tend to interact with their index fingers for a particular task while using their thumbs for other tasks. Some users tend to interact with slow fluid motions on specific tasks while using short "staccatto" motion on others. Table 2.4 shows a gesture example where the values on the Y-axis are almost constant. However, the X-axis values contain a high variation, which means that the movement or gesture occurred on the screen's horizontal axis. Also, pressure and area contain unique values constant between the same user and other users. Those differences may be used to detect malicious access (PATEL et al., 2016). Table 2.5 summarizes the most representative works on gesture recognition, using equal error rate (EER) as a metric to compare their results.

(SAE-BAE; MEMON; ISBISTER, 2012) introduced another authentication using multitouch gestures. The main idea of this paper was to use all the critical features in the hand geometry, creating a set of 22 gestures for authentication. (FENG et al., 2012) proposed to use touch data, adding an extra digital sensor in order to obtain pressure and acceleration, using false acceptance rate (FAR) and false rejection rate (FRR) to evaluate their results. (LUCA et al., 2012) introduced another technique, using password patterns with an additional security layer (transparent to the user). The authors conclude that gesture recognition has good potential for user authentication in real-world applications in those works.

Table 2.5 – Gesture recognition: research work summary

Work	Technique	EER
(NGUYEN; SAE-BAE; MEMON, 2017)	Draw PIN, PCA and DTW	4.84
(ROY; HALEVI; MEMON, 2014)	Hidden Markov Models(HMM)	0.0 - 5.0
(FENG et al., 2014)	1-NN and DTW	10.0
(ZHAO; FENG; SHI, 2013)	Images gray scale- z-score	6.3 - 15.4
(FRANK et al., 2013)	k-NN technique and SVM to classify	0.0 - 4.0
(LUCA et al., 2012)	DTW - compare distance between gestures	4.0
(SAE-BAE; MEMON; ISBISTER, 2012)	k-NN technique, DTW and Frechet distance	1.58
(FENG et al., 2012)	Decision tree, Random forest and Bayes net	FAR - 7.5 FRR - 8.0

(FRANK et al., 2013) proposed an encoding for gesture information in order to perform continuous authentication. In their encoding, they create a feature vector based on 30 behavioral features that can be extracted from the touch screen input of mobile devices and used to train an authentication model. These behavioral features contains: coordinates of the position of the finger during the stroke n, x_n and y_n; timestamp t_n; finger pressure p_n; area covered by the finger a_n; and orientation of the device o_{fn} and o_{pn}. In the latter case, the cellphone's orientation, the gestures can be different in portrait and landscape orientation, i.e., in the portrait orientation, gestures are shorter on the x-axis than on the y-axis. However, we have the opposite in landscape orientation, with rude gestures on the y-axis but large gestures on the x-axis. They evaluated k-Nearest Neighbors (kNN) and Support Vector Machines (SVM) as possible classifiers for authentication.

In addition, (ROY; HALEVI; MEMON, 2014) used the same data as (FRANK et al., 2013), but replaced the kNN or SVM classifiers by a Hidden Markov Model (HMM).

With those works (FRANK et al., 2013; ROY; HALEVI; MEMON, 2014), we can conclude that gesture sequences generated by one user are similar to each other, *provided the use case remains the same*, e.g., we analyze data on the same task/application. However, the similarity of each gesture sequence change if we change the application. That means that: the user's behavior is unique for each type of application, but data for the same user may vary widely across different tasks/applications. That complicates authentication considerably.

(ZHAO; FENG; SHI, 2013) proposed encoding the gesture information into a grayscale image in order to use image-classification techniques for authentication. That new type of image represented the movement of each user on a touchscreen, which is very different among users but similar for the same user. (ZHAO; FENG; SHI, 2013) uses six types of gestures to generate this type of image, such as coordinates, timestamps, and pressure — the most relevant introduced by (FRANK et al., 2013)— and also, cubic interpolation to normalize all of them. (ZHAO; FENG; SHI, 2013) uses three kinds of score metrics to compare all images — normalized cross-correlation, L1 norm, and L2 —

and chose L1 as the best metric.

The experiments were performed in controlled scenarios in all previous works, which may not reflect real-world usage. To solve this problem, (FENG et al., 2014) implemented a system of authentication in a cellphone's background environment, obtaining gestures in a more realistic scenario for continuous authentication - (FENG et al., 2014) use, in this case, a background app for collecting all gestures around apps in mobile devices -. (FENG et al., 2014) approach implements one gesture authentication by each application in the mobile devices, which means because the gestures for one application are pretty similar, but for another application, they are entirely different.

Finally, (NGUYEN; SAE-BAE; MEMON, 2017) proposed an alternative method for user authentication during the entering of PINs. The key idea is to let a user draw the PIN several times (e.g., Between 3 to 5 times) on a touch screen instead of typing. The system extracts from each sample gesture information such as position, pressure, and size of the touch area. The system uses principal component analysis (PCA) in the first phase to discriminate all drawings and dynamic time warping (DTW) for user authentication.

We choose to use the features established by (ZHAO; FENG; SHI, 2013) and (FRANK et al., 2013) such as position X, position Y, pressure, and area covered (the latter two being estimated, not measured). We choose the method of images proposed by (ZHAO; FENG; SHI, 2013), that encodes the data series into a gray-scale image, thus allowing to use architectures such as convolutional neural networks for the task of user authentication. We proposed to improve that scheme for continuous authentication by using transfer learning: our results appear in Section4.2.3. We later modified our design to deal with the gesture information directly, without the image encoding, allowing to implement the model as a LSTM instead of a CNN.

3 Proposed Architecture

This chapter describes our proposed solution based on a hardware design we called Mini Mind. We first summarize our design process and then give a detailed description of Mini Mind's units and the interface and protocol used to connect Mini Mind with a generic processor.

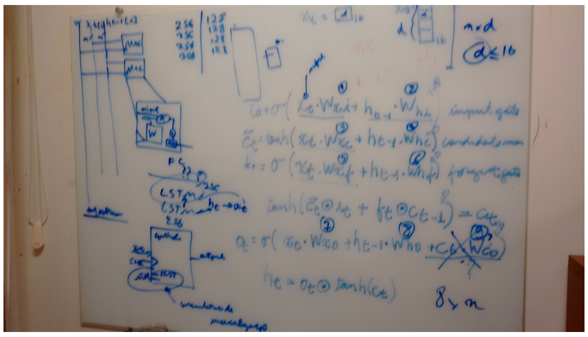
3.1 Methodology for Hardware and Software Design

Embedded systems design requires a rigorous methodology, which comprises problem analysis, algorithm design, system modeling and simulation, hardware design, and hardware deployment. State of the art on those methodologies has progressed steeply in the last years, especially in what concerns FPGA prototyping. However, there are still challenges to be overcome, depending on the type of application, since optimizing the algorithm to fit into a limited number of logic elements, embedded memory used, power consumption, and time constraints is far from obvious. Synchronization issues are also a source of challenges.

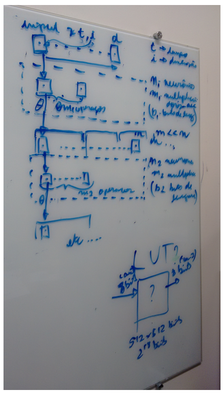
For Mini Mind, we took inspiration from the works of (BARKALOV; TITARENKO; MAZURKIEWICZ, 2019), (LI et al., 2019), and (VIDAL; CRUCES; ZURITA, 2014), since their hardware/software co-design, and their platform-independence was particularly useful to us. Using those tools, we could model the trade-off between network performance and available logic elements in hardware, which was the main limiting factor in our design.

Considering the hardware we had available (Altera Cyclone IV) during the design process, the original Mini Mind design changed. We attempted several alternatives for the expensive linear processing, which involves large matrix-vector multiplications, and non-linear activations, which involve approximating a transcendental function. When we committed to a pipelined design, synchronization and timing considerations became paramount. During that phase, in addition to the automated tool, we used free-hand drawings and storyboards to guide our design (Figures 3.1a and 3.1c).

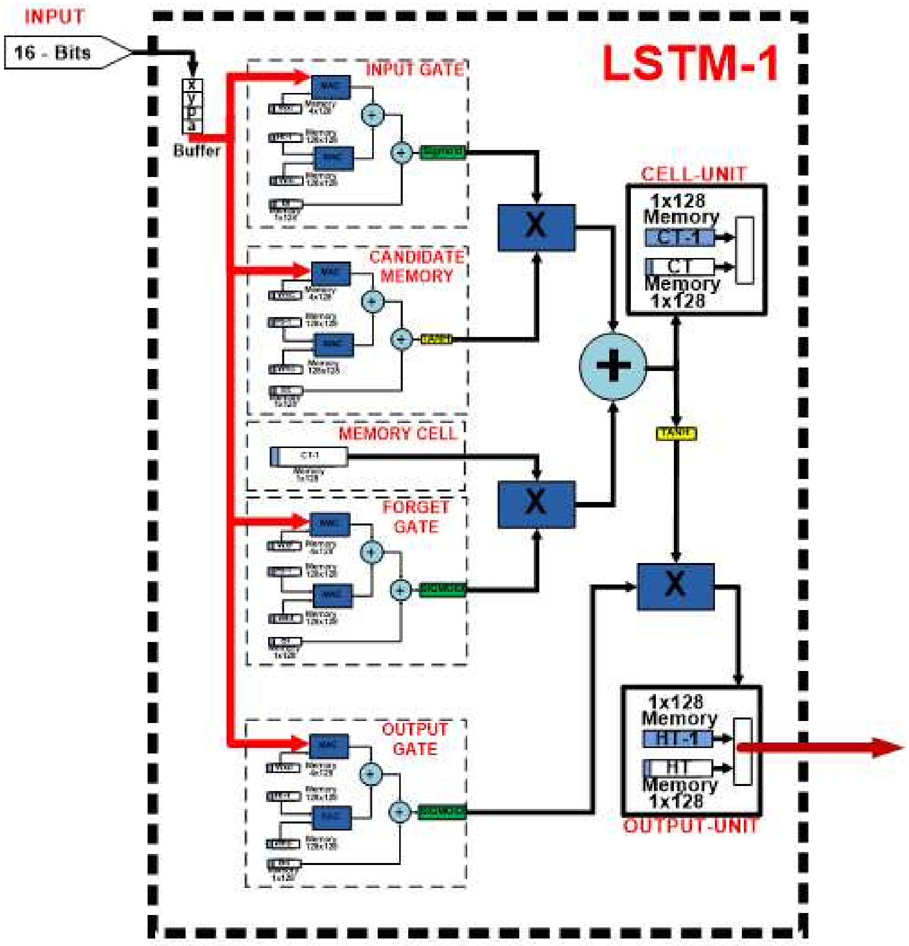
The significant advantage of mini mind is the management of internal memories implemented in the pipeline, maximizing the throughput using a loop cycle that we had to optimize by hand, using low-level tools/language (e.g., VHDL writing). Solving synchronization issues took a significant part of the development process.



(a) First Scratch - Estimation of computational resources using all LSTM equations *input gate* (2.2), *forget gate* (2.3), *output gate* (2.4), *candidate memory* (2.5), *memory cell* (2.6) and *output gate* (2.7).



(b) Second Scratch - Mini Mind Design - Pipeline approach



(c) Final Design from second Scratch- Pipeline approach

Figure 3.1 – Mini Mind Design Process: figures (a) and (b) are samples from the actual design sessions during prototyping, which used extensive collaborative drawing and redrawing on whiteboard. Figure (c) is a refined drawing of the resulting architecture.

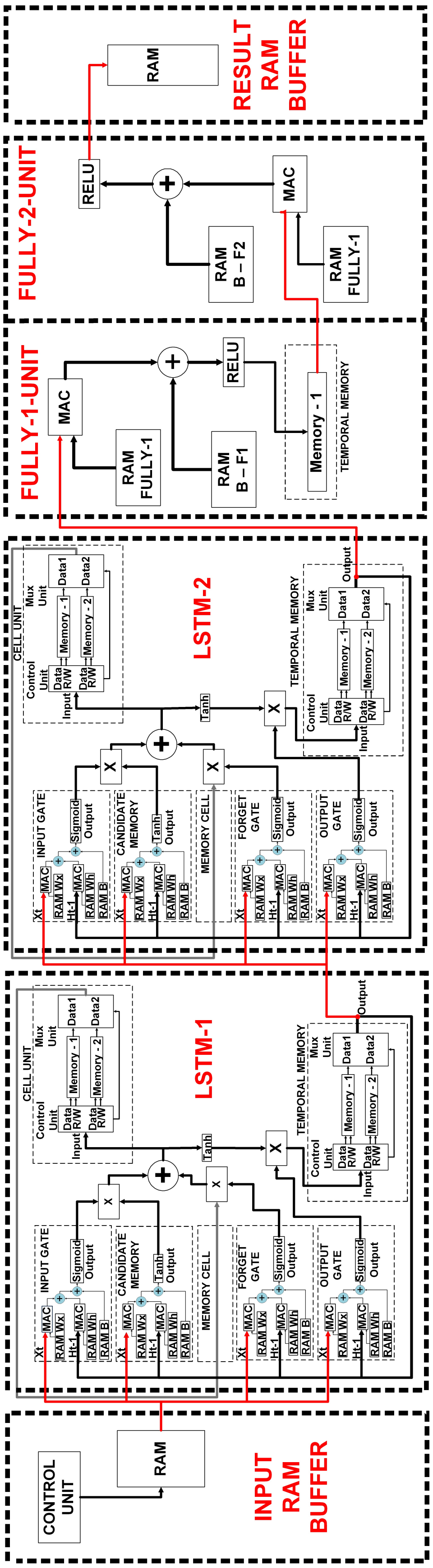
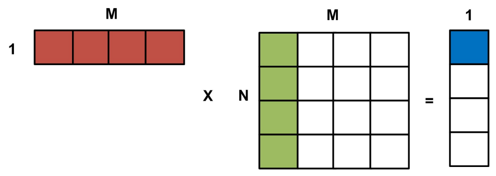


Figure 3.2 – Mini Mind architecture

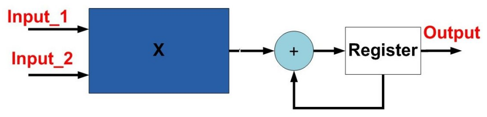
3.1.1 Proposed Mini Mind Architecture

Mini Mind is a hardware accelerator based on the work of (CHANG; CULURCIELLO, 2017). It has two LSTM layers and two fully-connected layers, each layer with 128 units. For todays' standards, that is a relatively shallow network, but it still can be used both as a discriminant and a generative model for many practical tasks. This work will validate the design for continuous user authentication using gestures. The parameters of our architecture will be described in detail in Section 4.

For the LSTM layers, the hardware implementation has to implement equations: *input gate* (2.2), *forget gate* (2.3), *output gate* (2.4), *candidate memory* (2.5), *memory cell* (2.6) and *output gate* (2.7). In this context, (CHANG; CULURCIELLO, 2017) proposed to implement each unit separately for LSTM layer using a MAC unit. For a better explain, Figures 3.3a and 3.3b show that approach, which considerably reduces the need for multipliers, but increases the memory used to store the weights. The proposed process performs multiplication and accumulation for all units, and stores the results in RAM. In the speed *vs.* hardware size compromise, that design greatly emphasizes the latter: the basic units are reused and there is no need for additional elements, but the whole process is very time-consuming, because each network inference requires sequential computation of the network layers, with intermediate storage and retrieving of partial results in RAM.



(a) Matrix-vector Multiplication



(b) Multiply–Accumulate Operation (MAC)

Figure 3.3 – Matrix Multiplication using MAC unit

Therefore, our model seeks a more balanced speed-*vs.*-hardware size compromise than (CHANG; CULURCIELLO, 2017), by employing pipeline parallelism on the network layers. That requires each layer to have its independent hardware components but can significantly accelerate inference since the network can parallelize the processing of its sequential input using the pipeline. That scheme also accelerates the matrix-vector multiplications, processing all the gates and memories of a unit in parallel.

Managing numerical precision was another critical design decision. We followed (CHANG; CULURCIELLO, 2017) and employed fixed-point 16-bit numbers, with 8 bits for an integral part and 8 bits for the fraction ("Q8.8" for short).

3.2 Mini Mind Design

In this section, we detail Mini Mind's design, describing our choices for the hardware implementation, the optimizations we made for speeding up the design, the synchronization issues, and the numerical approximations.

3.2.1 Resources Estimation

Before carrying out the design, we created a model to estimate the needed elements in hardware using the number of units of LSTM layers and the number of multiplications for each layer (MAC units) and memories (Table 3.1). Those estimations helped us optimize the design while ensuring it would fit the available hardware.

In both LSTM and fully-connected layers, the most critical operations are linear matrix-vector multiplications, where the matrices store the model's parameters, and the vectors are the inputs to that layer (coming from the previous layer, or the user, in the case of the first layer). The other essential operations are the non-linear activation functions, but those are "simpler" since they are applied element-wise to the vectors and have no learnable parameters.

Table 3.1 shows our estimations for two sizes of the LSTM layers (128 and 256). In that table, we only rely on the number of basic elements necessary to implement such a network. In our case, the most critical factors are the MACs and multipliers units. In the most basic model of this table, the use of a single multiplier for each unit, as shown in equations 2.2 , 2.3, 2.4, 2.5, 2.6, 2.7. The *x2*, *x4* are MAC's units required to implement that factor. As previously stated, in the Mini Mind architecture, increasing the speed factor is directly proportional to the number of multipliers involved, dividing the task and increasing the final throughput.

That estimate depends on how many essential elements (lookup tables) will be necessary to implement, the number of bits (system precision), and the synthesis tool's optimization. In addition, the FPGA board's capacity since the number of Macs and lookup table elements varies depending on the FPGA board and manufacturer. In the memory factor, the memory value required does not change (Depending on the software tool, memory values could change depending on the size and optimizations of software, but for our analysis, we have not considered this variation). The memory of the network weights used to implement is fixed in any situation. We used an estimated LSTM network with 128 and 256 neurons in the table. In this time case, the estimate is not a linear factor because adding adders is required after the multiplication process. In this case, the total sum of each adder in the cascade is considered with an estimated processing time of a clock cycle.

Table 3.1 reflects only the behavior of multipliers and memories for the LSTM

Table 3.1 – Estimation of Mini Mind architecture. The units counted are DSPs and multipliers. % over the total available in Altera Cyclone IV. Time in ms.

	Design	**Units**	**% used**	**Memory**	**% used**	**Time**
1	Sequential – 256	68	9.4	13 795 328	208	49
2	Sequential – 256 (x2)	104	14.4	13 795 328	208	27
3	Sequential – 256 (x4)	176	24.4	13 795 328	208	16
4	Sequential – 128	68	9.4	3 473 408	52	13
5	Sequential – 128 (x2)	76	10.6	3 473 520	52	8
6	Sequential – 128 (x4)	176	24.4	3 473 408	52	5

layers, and it was not made for fully connected layers because including additional hardware would not have a significant impact on the final throughput. Another reason for the bottleneck of the multiplication process in LSTM layers is the multiplication of basic units, and that is why adding extra hardware (multipliers) in this process does not substantially increase our throughput. Also, the fully connected architecture already trained can be replaced with one multiplier, one adder, one activation function (Relu in our case), and a memory block (Depending on the size of our layer - Number of neurons in the layer), simplifying our hardware architecture (It only need one MAC unit to represent an all fully connected layer operation).

In LSTM networks, it is not always possible from a hardware point of view the use external memory to partially store the results from the multipliers depending on the size of the architecture. An LSTM architecture with 256 neurons could not be deployed to our Cyclone IV boards because of memory limitations. That is why many authors proposed optimizing or reusing the lower layers of deep learning architecture to implement more complex architectures on even more limited hardware. In our context, this approach is well received when we use CNN architectures when 90% of all parameters are around the firsts layers. However, even with this approach, some FPGA boards cannot implement a CNN network in a pipeline scheme because of the number of MAC units and memory required by the whole network. However, in the case of LSTM architecture, it is possible to implement in a pipeline all layers using the most straightforward hardware architecture, one multiplier per basic unit.

3.2.2 Balancing Design Complexity and Numerical Accuracy

Managing numerical precision throughout the design is critical for implementing neural networks in hardware. In particular, we approximate all non-linear functions with a linear implementation, obtaining again in hardware implementations. In our case, we are trying to implement the logistic sigmoid and the hyperbolic tangent activation functions, both of which are transcendental and have to be approximated.

A single polynomial can approximate a function, but good approximations require high-order polynomials, presenting computational performance and numerical stability issues. An alternative is to approximate them *piecewise*, in which case even low-order polynomials, like linear functions, may provide reasonable approximations, provided there are enough "pieces". The number of pieces is directly proportional to the number of the elements in hardware required and correlates with the approximation obtained.

Before committing to a given approximation in our hardware, we ran several simulations in software (Figures 3.4 and 3.5). We fit piecewise linear approximations for sigmoid and tangent functions using Matlab. In these graphs, we calculate the root mean square to calculate the error between our approach line and the curve of the non-linear function — considering the base result from Matlab. In this case, 35 simulations were carried out in which it could be seen that as the use of points increases, the error tends to decrease considerably. We vary the number of interpolation points N from 2 until 36, resulting, in each case, in $N-1$ pieces. For each N, we calculated the root mean square (RMS) error between the obtained approximation and the reference IEEE 754 implementation of those functions available in the software. As N increases, our approximation gets very close to the reference implementation, but large values will need more hardware elements in the final design.

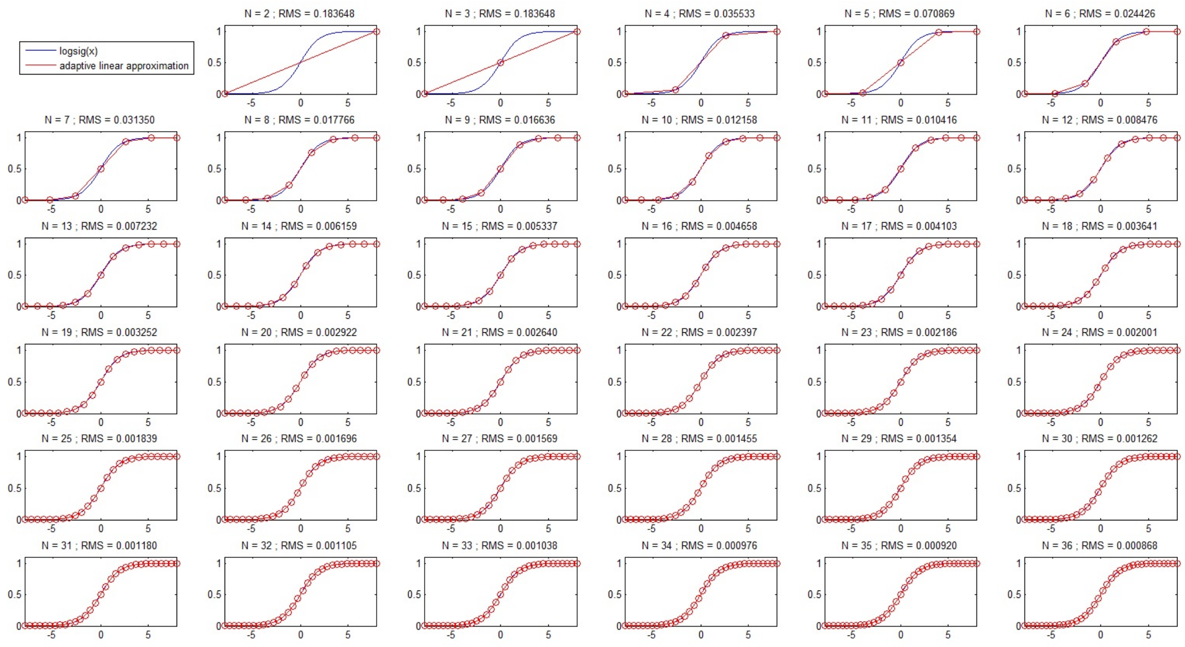


Figure 3.4 – Logistic Sigmoid Approximation. Using piecewise function to approximate the nonlinear function, where N is the number of points used in each graph.

Using preliminary experiments, as shown in Figure 3.4, we chose $N = 4$ for the logistic sigmoid, corresponding to an activation function called "hard sigmoid" in deep learning literature. Hard sigmoid is the default sigmoid function in Keras, which is faster to compute than the full-precision sigmoid activation. We used the hard sigmoid to perform *both* the training and the inference phase of the model.

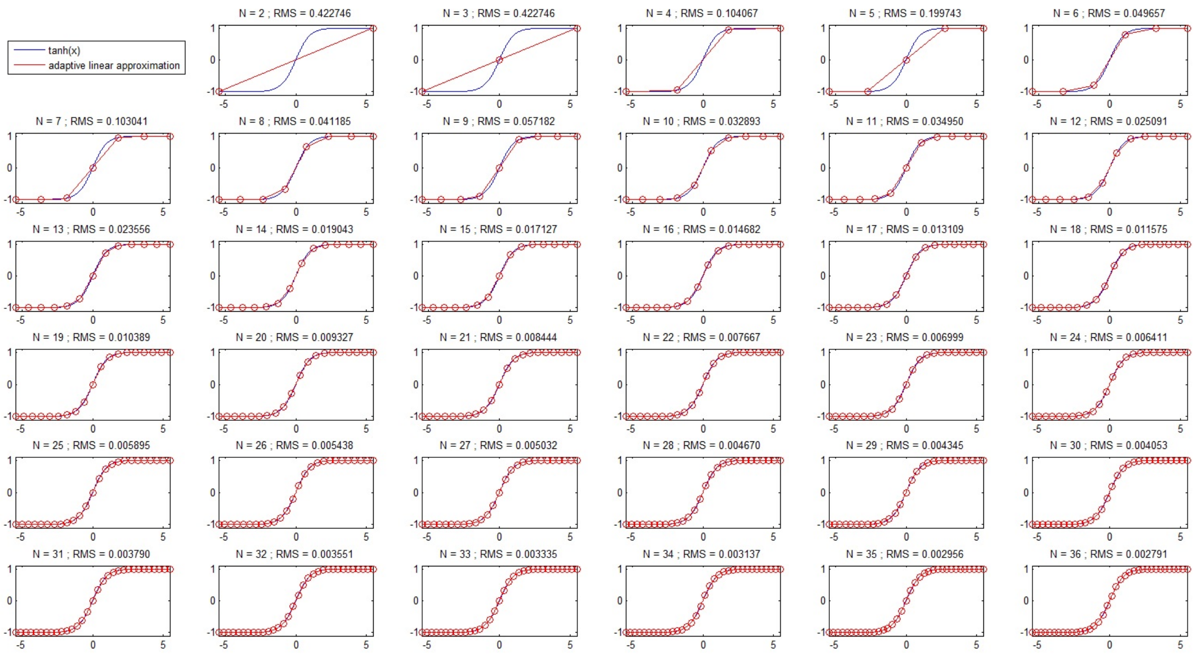


Figure 3.5 – TanH Approximation. Using piecewise function to approximate the nonlinear function, where N is the number of points used in each graph.

For the hyperbolic tangent, as shown in Figure 3.5, we chose $N = 12$, which we considered a good compromise between hardware parsimony and numerical approximation, especially considering the intrinsic limitations of the Q8.8 fixed-point employed in our design.

3.2.3 Implementation Details

As mentioned, we implemented Mini Mind architecture in an Altera Cyclone IV FPGA. Specifically, we employed a prototyping/development board Altera Cyclone IV GX, model EP4CGX150DF31.

Mini mind architecture optimizes matrix-vector multiplication in the LSTM layers by using temporary memory to interconnect them in a pipeline (Figure 3.2. Inside each layer, the datapath computes all gates and memories in parallel).

To allow testing, we implemented an additional layer able to communicate with a Nios II processor (A generic processor to emulate a mobile device for sending data and executing an instruction at each clock cycle), as shown in Figure 3.6. Using the Nios II processor simulates a realistic scenario where Mini Mind can operate and communicate.

3.2.4 Input Data Sequence

Some choices we took for Mini mind were motivated by the application target we chose, gesture recognition, and will be better motivated in the next chapter, in

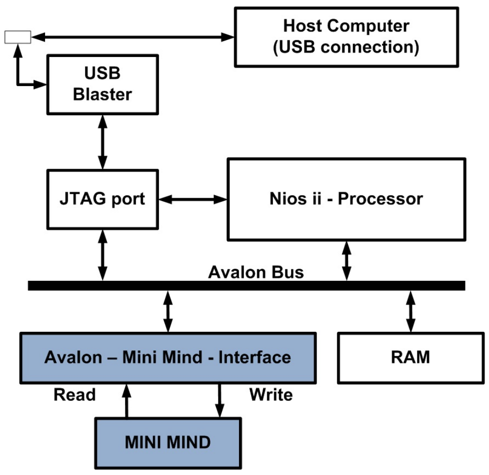
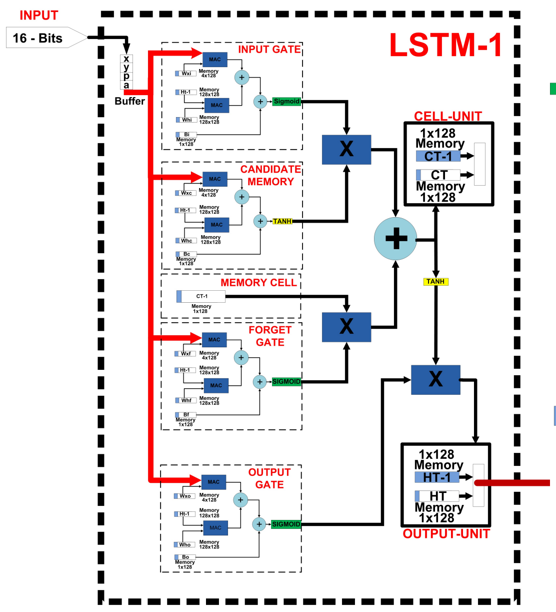


Figure 3.6 – Mini Mind architecture with avalon interface and Nios-II Processor

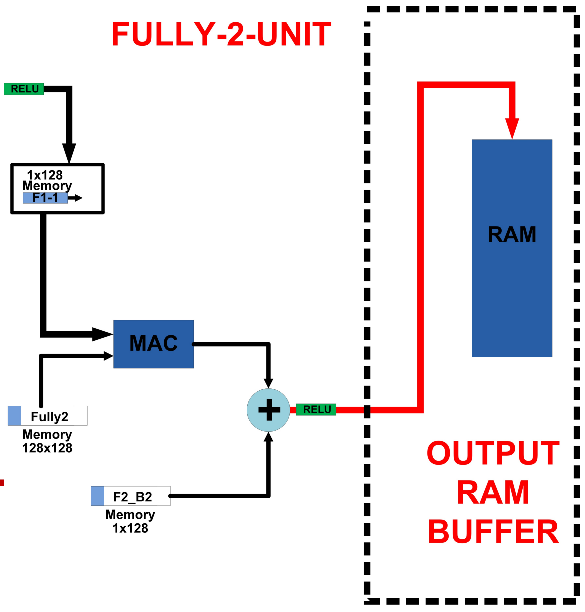
Section 4.1.0.2. For the moment, we ask the reader to accept that the input to our network is a vector of size 32×4.

3.2.5 Input and Output

We used a memory buffer for the input and output phases. Those units have a control unit based on Algorithm 1, with the primary purpose to synchronize all data flow in all gates of the LSTM layers, shown in red in Figure 3.7a. The output memory saves the results after the entire data sequence is processed, guided by the control unit, as shown in Figure 3.7b. A library memory element provided by Altera — RAM: Single-port (ALTERA, 2014) — with 128 words is used for input and output.



(a) Input data flow in Mini Mind



(b) Output data flow in Mini Mind

Figure 3.7 – Input/Output Units in Mini Mind

3.2.6 Basic Mini Mind Unit

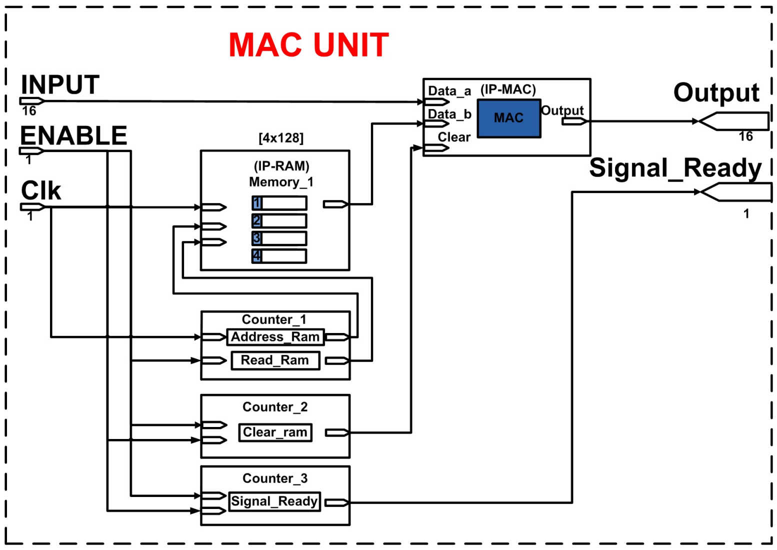


Figure 3.8 – Basic Mini Mind Unit - Gate

3.2.6.1 MAC unit

To perform the vector–matrix multiplications (Figure 3.3a) in the LSTM layers, we developed a basic unit using MACs (Figure 3.3b). Those MAC units are used to implement each gate (2.2), (2.3), (2.4), and (2.5) in parallel. To synchronize all blocks units to perform matrix multiplication (Figure 3.9b), we use counter units, based on Algorithm 1, which serve as pointers to address RAM. Using IP library elements, we could read one value per clock cycle. In Algorithm 1, n represents the number of neurons in the design (In our case, the counter is to 128). The values *Read* and *Enable* are for enabling a reading process and MAC multiplication. Moreover, the *Delay* value represents the time necessary for a temporal memory writing the last value when a complete matrix multiplication is completed.

Algorithm 1 Unit Basic of Multiplication

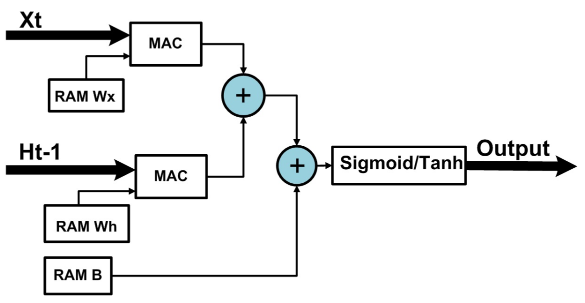
```
procedure PROCESS MAC AND RAM(Clock event)
    if $Counter_1 <= n$ then
        if $Address = n$ then
            $Address = 0$
            $Counter = Counter + 1$
        else
            $Address = Address + 1$
            $Read = 1$
            $Enable = 1$
        end if
    end if
    if $Counter_1 > n$ then                                        ▷ Extra delay
        $Address = initvalue$
        $Read = 0$
        $Enable = 0$
        if $Delay = m$ then
            $Counter = 0$
            $Delay = 0$
        else
            $Delay = Delay + 1$
        end if
    end if
end procedure
```

3.2.6.2 Gate unit

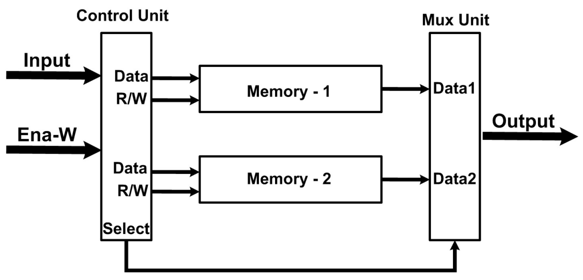
We created a generic hardware gate implementation, which we could use to implement all gates in an LSTM layer: input, candidate, forget, and output. Figure 3.9a shows the elements in the gate unit:

- Two MAC units, which can perform multiplication of two numbers of 16 bits, resulting in 32 bits (which we truncate into 16 bits so that we can iterate the process).
- Three memory units that store all gate parameters: W_x, W_h and b_n.
- Two 16-bit adders.
- One block of approximating the (sigmoid or hyperbolic tangent, depending on the gate) activation function.
- One register for the result.

Equation 3.1 shows *hard sigmoid* we employed in the place of the logistic sigmoid (as explained in Section 3.2.2).



(a) Gate Unit Configuration



(b) Temporal Memory Unit

Figure 3.9 – Basic Units in Mini Mind

$$HardSigmod(x) = \begin{cases} 1 & \text{x} > 2.5 \\ 0.25 * x + 0.5 & \text{Otherwise} \\ 0 & \text{x} < \text{-2.5} \end{cases} \tag{3.1}$$

For the hyperbolic tangent function, we employ 12 linear polynomials $y = a \times x + b$. A single MAC unit is employed, with parameters a and b selected according to the value of x by the control unit.

We only use one unit of multiplier, adder, and control in our case. The values a and b are updated by a control unit. A control unit depends on the value of x, selecting the best line (Between 1 to 12) to perform an interpolation of x.

3.2.7 Temporal Unit Memory

The temporal memory module allows storing partial matrix multiplication in vectors, as shown in Figure 3.9b. One of the essential advantages of Mini Mind is using these temporary memories with control units, which allows us to use a pipeline architecture. We need two temporal memories to provide all gates, both the previous output h_{t-1} (which is used as an input to the gates) and the space to store the current output h_t.

The temporal memory design, containing:

- Two RAMs of 128 words.
- One control unit, which can select the correct memory for writing and reading.
- One MUX unit.

In all this process, the control unit selects which memory to read and write, i.e., the two temporal memories' roles are switched when we progress from one timestep to the next one, i.e., the memory used as output buffer (for h_t) at time step t will be used as the input buffer (h_{t-1} at timestep $t + 1$, and vice-versa. That scheme, coordinated by the control unit, allows efficient memory use, with the storage needed for a single timestep and zero copying of data between buffers. With control and temporal units, we eliminate the need to use large-size memories to store partial results by each LSTM layer (Limited by the size of deep learning architecture to store all weights in the memory slots in FPGAs).

3.2.8 Pipeline

One of the advantages of Mini Mind is the use of pipelines through the LSTM layers, using small units, and reusing RAMs to store the network parameters and partial results.A single LSTM layer has, for each gate, parameters W_x, W_h, and b_n (Figure 3.2). With small memories (on-chip block memory), it is possible to execute one multiplication per clock cycle, parallelize all gates units, and use a pipeline architecture for all layers. For that purpose, we use memories such as:

- *Memories of 128 words*: For input and output buffer, bias term, and temporal units. Mini Mind contains 20 units.
- *Memories of 512 words*: The weights with direct operation with input vector in the first layer of LSTM. Mini Mind contains four units.
- *Memories of 16384 words*: For the rest of the parameters, Mini Mind contains ten units.

Those memory sizes, the number of units corresponding to the number of neurons, and the number of hidden layers present in Mini Mind. As explained before, all memories store 16-bit words of numbers in Q8.8.

3.3 Nios II Interconnection

Nios II is a 32-bit embedded-processor architecture explicitly designed for the Altera family of field-programmable gate array integrated circuits. Nios II processor can interconnect all components in the Altera Cyclone IV FPGA chip to form a complete system (EKAS; JENTZ, 2003). Those elements use a bus data interconnector called Avalon Switch Fabric. A host computer can control the Nios II processor (using the JTAG Debug module) to perform operations such as uploading programs into its memory or collecting data. (INTEL, 2016; ALTERA, 2011).

Figure 3.6 shows a schematic design of Mini Mind interconnection with the Nios II processor using the Avalon bus. To interconnect Nios ii with Mini mind, it was necessary to design a new component that transforms the Nios II preventive instructions into 32 bits for a 16-bit system with Mini Mind.

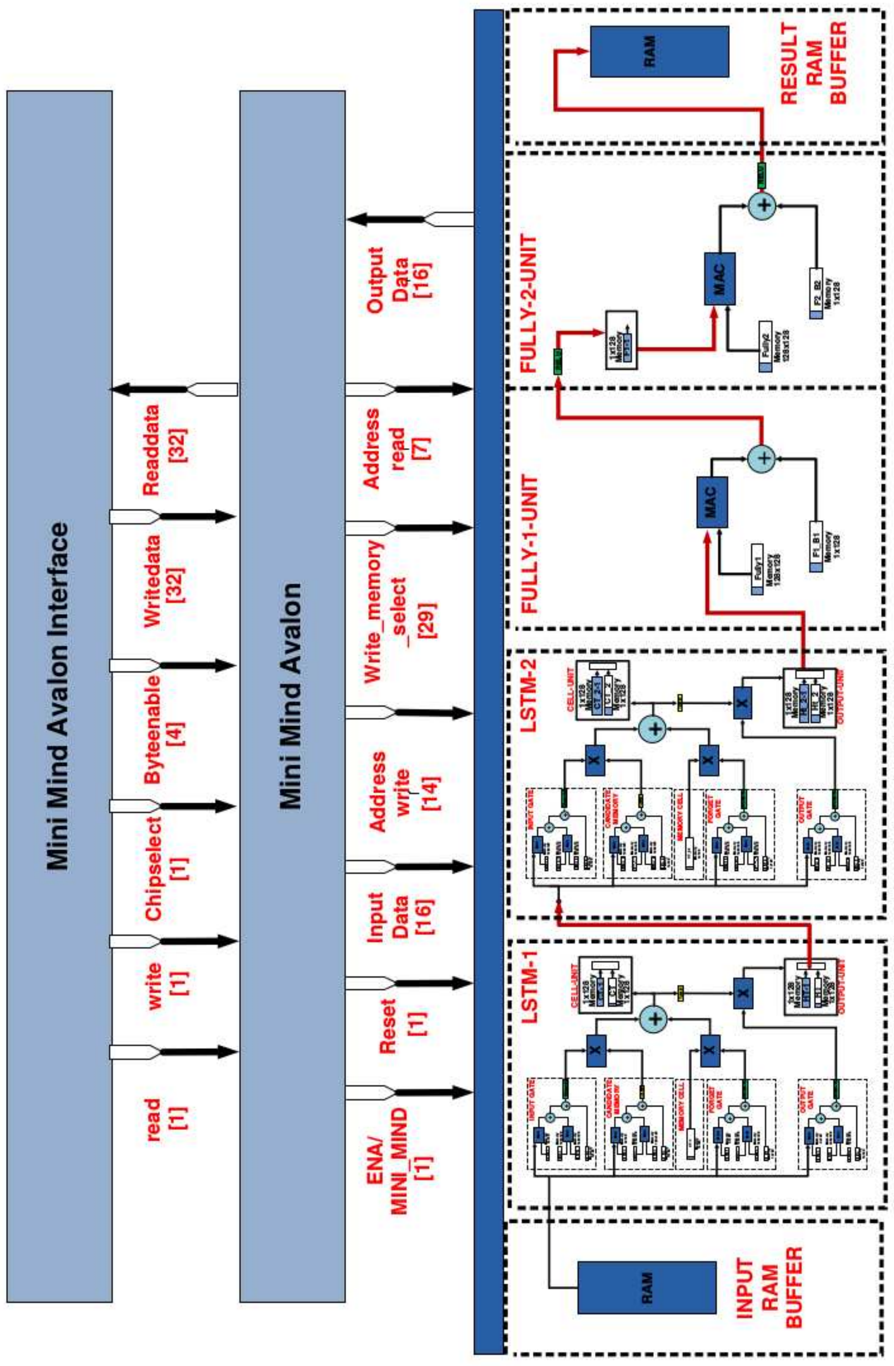


Figure 3.10 – Mini Mind architecture With Nios II

The Nios II uses 32 bits per instruction so that the Mini Mind interface can translate a single instruction by a processor into instruction for Mini Mind hardware, for example: save data in the input buffer or read data from the result ram buffer. The instructions for Mini Mind are:

- Input data (15 down to 0): The data in 16 bits.
- Address memory (22 down to 16): Used to indicate an address in input memory buffer
- Instruction (25 down to 23): Instruction op-code, as indicated below:
  - *Address write (001)*: Enable writes mode for the address input buffer.
  - *No address write (010)*: Disable writes mode for the address input buffer.
  - *Reset all variables (011)*: Reset all memory counters and temporal variables.
  - *Enable mini mind (100)*: Start process.
  - *Disable mini mind (101)*: Stop process.
  - *Write weight memories (110)*: Enable update mode for the parameters in the memory units.
  - *Read-only weight memories (111)*: Disable update mode for the parameters in the memory units.
- Select Memory (30 downto 26): Allows selecting an specific parameter memory (W_x, W_h and b) in Mini Mind to update (there are 29 parameter memory units, besides the input/output *data* memories, which are not concerned here).

The Nios II processor was essential for fast data flow. The processor allows moving data from memory to memory while keeping the design of the units reasonably independent.

3.4 Discussion

The development and design of Mini Mind posed five significant challenges. First is implementing the LSTM layers, which we tackled by splitting the architecture into basic units. The numerical properties of the units — mainly the non-linear activation functions — were first simulated in software (as explained in Sections3.2.2). Although the isolated design of the basic units was relatively simple, the main challenge was in their integration, where limitations due to the timing of the components became critical.

Second, a significant challenge was balancing the number of MAC units to optimize the model's time performance while still fitting it into the available hardware. The model we used as a baseline was optimized only for minimal hardware usage, sacrificing speed. Instead, we chose to employ more logic units to allow the parallel computation of the gates and pipeline the execution of the layers while still using a reasonable number of units.

Third, carefully managing memory storage by carefully managing numerical precision and by judiciously distributing storage of the weights, input, and output buffers. The use of double-buffering for input/output between timesteps allowed us to move from one timestep to the next while copying any data and keeping the pipeline flowing.

Fourth, the interconnection among LSTM layers. The interconnection between the basic modules and the layers needs to be synchronous from beginning to end of all processes. This module was necessary to include some extra control to perform this type of synchronization because a slight variation in some modules can significantly impact the final value during the inference process. Also, an LSTM architecture contains a recursion module; that is why the error among modules tends to grow exponentially in each processing cycle.

Finally, the interaction with a generic processor (NIOS II) was essential to run the Mini Mind in real-time, providing the ability to initialize its weights, feed its input and collect its output.

4 Experiments and Results

In this chapter, we present our experimental results. We will showcase our selected application — continuous authentication of users on mobile devices.

First, we will present a baseline solution, which was implemented only in software, as an evolution over the works of (ZHAO; FENG; SHI, 2013) and (FRANK et al., 2013). While they encoded their gesture data into grayscale images, we encoded ours into RGB images, allowing more information. The images are then classified using deep convolutional neural networks, which we implemented in software. Those are the earliest results obtained in this thesis and served as a baseline for the remaining results. However, as we advanced in the hardware design, we decided that convolutional networks were way too expensive for the hardware we intended and moved towards recurrent networks.

The solution based on recurrent networks in shown next. We implemented it twice, using software (in Python/Keras) and hardware (in Mini Mind, which we developed and explained in the previous chapter). The software design also served as a baseline for the hardware design.

We will start the chapter by reintroducing continuous authentication and explaining the available data for our tests. We will then show the results we obtained for each of our implementations.

4.1 Gesture Recognition for Continuous Authentication

As discussed in Section 2.3, gesture recognition for continuous authentication is a relatively novel idea, with limited literature and even fewer works exploiting deep learning, either with CNNs or LSTMs.

4.1.0.1 Public Dataset

The first dataset we employed in our experiments was the publicly available Touchalytics dataset[1], developed by (FRANK et al., 2013). It contains the raw touch data (Complete sequence of a single gesture) of 41 users in scenarios like reading Wikipedia articles or playing an image comparison game.

In the dataset created by (FRANK et al., 2013), we found that some gestures have different sizes and lengths. The length of the gesture depends on how much time the user interacts with the touch screen using their fingers. In this context, it is recommended

[1] Available in http://www.mariofrank.net/touchalytics/data.zip

to use a normalization process to leave all gestures the same size. In our case, we use linear normalization, excluding gestures with lengths Less than six-time sequences. In this process, we take off at least 15.16% of all gestures in that dataset. Table 4.1 shows a summary of the rest of the gestures of each user class.

In this dataset, we selected random samples to authenticate users, and we separated for each user 50% of the samples for training, 25% for validation, and the rest for a test.

Table 4.1 – Number of Gestures per User

U-1	**U-2**	**U-3**	**U-4**	**U-5**	**U-6**	**U-7**	**U-8**
353	1012	651	219	381	532	369	437
U-9	**U-10**	**U-11**	**U-12**	**U-13**	**U-14**	**U-15**	**U-16**
445	374	434	270	284	503	420	362
U-17	**U-18**	**U-19**	**U-20**	**U-21**	**U-22**	**U-23**	**U-24**
670	455	290	255	692	322	590	359
U-25	**U-26**	**U-27**	**U-28**	**U-29**	**U-30**	**U-31**	**U-32**
377	185	561	550	341	203	311	296
U-33	**U-34**	**U-35**	**U-36**	**U-37**	**U-38**	**U-39**	**U-40**
581	593	1021	384	381	671	364	292
U-41							
172							

4.1.0.2 Private Dataset

The private dataset was the RECOD Gestures dataset, collected at RECOD–IC and LCA-FEEC laboratories. In that dataset, the users have generated two types of gestures (Flick left and flicked right) while viewing images in a simulated photo gallery. For this purpose, we created a data collection application where each user performed the task of browsing a gallery of approximately 50 photos with a neutral subject (views of the buildings of the School of Electrical Engineering and of the Computing Institute of UNICAMP). While the users performed that task in the foreground, a data collection process ran in the background, collecting the position x-y, pressure, area, timestamp, or any interaction with the touchscreen.

The dataset has the gestures of 40 volunteer users who continuously interacted with their cellphone until 40 interactions were collected.

In the same case of the other dataset, we also selected random samples to authenticate users, and approximately, we separated each user, 50% for training, 25% for validation, and the rest for test data.

4.2 First Solution: Image Encoding and Deep CNN

Based on our case of study gesture recognition, we based on the essential features used by (FRANK et al., 2013) and (ZHAO; FENG; SHI, 2013) such as position X, position Y, pressure, and area covered [2].

$$S_n = (T_n, X_n, Y_n, P_n, A_n),\ n = \{1, 2, 3..n\} \tag{4.1}$$

The dataset used to perform our experiments was developed by (FRANK et al., 2013), it has 21174 gestures, and in the function of an application, the sizes may have a different time sequence. To normalize, we used linear interpolation because it required fewer operations in hardware and also decided to standardize all of them into a fixed size time sequence (e.g., 32).

The value of 32 for gesture normalization was selected for two reasons: the first, the CIFAR architecture using an input image of 32x32, the first software solution we proposed in this work. Furthermore, changing the value of gesture normalization does not improve the result of user authentication at the end.

The rest results of how this approximation was selected will be shown in the experiments and results in Section 4

Our first implementation extended the works of (ZHAO; FENG; SHI, 2013) and (FRANK et al., 2013), encoding gesture information into images we called GTGF-RGB, which use all three channels to represent information such as pressure, finger area, position x, and y with the equations in section 4.2.1.

We used a pre-trained deep network that has been trained on a large image dataset (CIFAR-10) as a feature extractor. The networks receive the encoded gesture images, and output feature vectors used to train our classifier. The aim, at the time, was to have a low-power, low-memory implementation that could be deployed in an Android OS phone model LG X Power K220.

We will first describe our scheme using RGB images for continuous authentication and then contrast them with the grayscale GTGF Images proposed by (ZHAO; FENG; SHI, 2013) and (FRANK et al., 2013).

4.2.1 Image Generation

In our GTGF-RGB implementation, each image captures the following gesture information:

- Position: X_n,Y_n

[2] In case of area covered and pressure are values estimated, not measured

- Pressure (calculated): P_n
- Timestamp: T_n
- Area of the finger on the screen: A_n
- Orientation of the cellphone" Or

That information is the most critical of each gesture (Equation (4.2)). Before converting a gesture into an image, it is necessary to normalize the size of each gesture. We use linear interpolation for that purpose, using the size of 32 for all pressure values (Equation 4.5) and 33 points for position values (Equations 4.3 and 4.4) because the use of deltas ($\Delta X_n \quad \Delta Y_n$) require an extra value, generating images of 32x32 pixels. We did preliminary experiments with different time-sequence lengths without significant improvement in the result.

$$S_n = (X_n, Y_n, T_n, P_n, A_n, O) \;\; n \; \epsilon \; 1, 2, ...n \tag{4.2}$$

$$I_n^p = \left[\frac{I_m \; * \; U_x \; - \; \Delta X_n}{U_x} \right] \;\; ; \; \Delta X_n = X_{n+1} - X_n \tag{4.3}$$

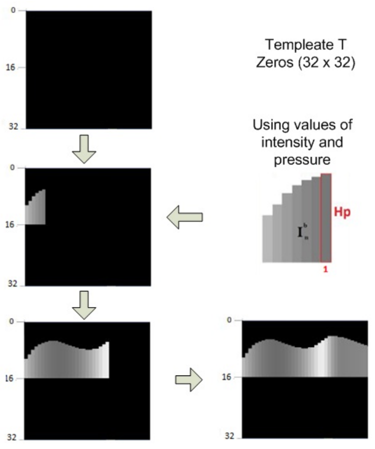
$$I_n^b = \left[\frac{I_m \; * \; U_y \; - \; \Delta Y_n}{U_y} \right] \;\; ; \; \Delta Y_n = Y_{n+1} - Y_n \tag{4.4}$$

$$H_n^p = \left[\frac{H_c \; * \; P_n}{L_p} \right] \tag{4.5}$$

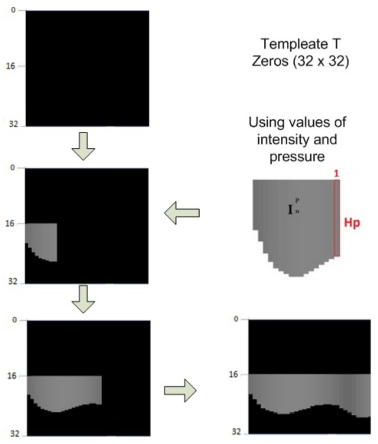
The variables U_x, U_y and L_p are validated with the Touchalytics training set to guarantee intensity values in the range of 0 to 255, in the case of L_p in the range of 0 to 32 (size of the image). The values I_m and H_c are 128 and 16 (half of the maximum values of intensity and image), respectively.

Figure 4.1 shows a process for converting gestures into an images. First, we create a zero-template (size of 32x32), then we divide into two parts: Upper and lower, the middle part represent an initial point for intensity values. Equation 4.3 calculates an intensity value of upper part corresponding a X values (X-Position value of gesture), each value represent one single point of intensity. Equation 4.5 calculates a height value of each intensity as shown in Figure 4.1a. We used the same process in lower part corresponding a Y values (Y-Position value of gesture) with the Equations 4.4, 4.5 in Figure 4.1b. Finally, we combine both parts generating the image based on gestures in Figure 4.1c.

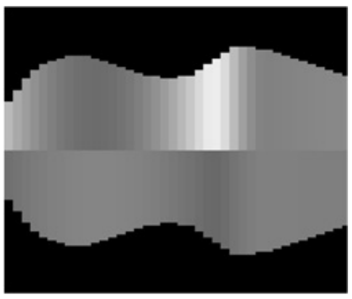
(ZHAO; FENG; SHI, 2013) method only uses one single channel per image to represent a little information of gesture (Position and pressure). However, in (FRANK et al., 2013) method, they use several methods adding more information of each gesture to increase accuracy. In this context, We modify the image GTGF proposed by Zhao



(a) Gesture conversion - Upper part



(b) Gesture conversion - Lower part



(c) Image generated by gestures

Figure 4.1 – Gesture convert into an Image

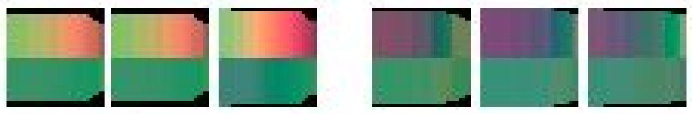
adding more channels (RGB) to represent more information of each gestures (area of the finger and orientation of the smartphone). We create Equations 4.6,4.7 (Based on the Equations 4.3,4.4,4.5).

$$G_{un} = \left[\frac{X_n}{Max_x}\right] \; ; \; G_{ln} = \left[\frac{Y_n}{Max_y}\right] \tag{4.6}$$

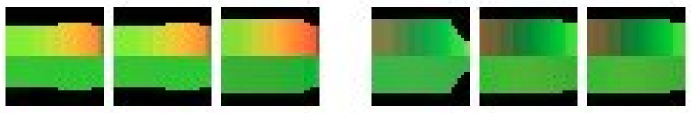
$$A_n^p = \left[\frac{A_n}{L_a}\right] \tag{4.7}$$

The values Max_x and Max_y in Equation 4.6 (G-channel) depend of the orientation of the phone. If the mobile is in portrait orientation, the maximum values are 1080 and 1920 for x and y. However, If the mobile is in landscape orientation, the maximum values are 1920 and 1080 for x and y.

We consider those two new Equations (4.6,4.7) because some parts in the touchscreen are very characteristics for each user and they are possible to assign intensity values of each part of the screen, representing in a better way gestures movements. The value L_a in Equation 4.7 (B-channel) is cross-validated with the training set to guaranty intensity values between 0 to 255.



(a) User-01 gestures 1-2-3 (move for right), gestures 4-5-6 (move for left)



(b) User-02 gestures 1-2-3 (move for right), gestures 4-5-6 (move for left)

Figure 4.2 – Images RGB based on gestures

Figure 4.2 shows an RGB image based on gesture, a Figure 4.2a and Figure 4.2b are gestures from different users. The gestures from the same user are similar, but if we compare the same type of gestures between other users are different.

4.2.2 Authenticating Users using GTGF-RGB

The primary purpose of generating images RGB based on gestures was to use Convolutional Neural Networks in the authentication process. Therefore, we used the Cifar-10 pre-trained architecture trained in Cifar dataset, as shown in Figure 4.4, as a principal architect for this purpose. We applied the transfer learning technique(CARVALHO, 2015)(PAN; YANG, 2010), removing the last fully connected layer and replacing it with the Support Vector Machine (SVM) classifier at the end.

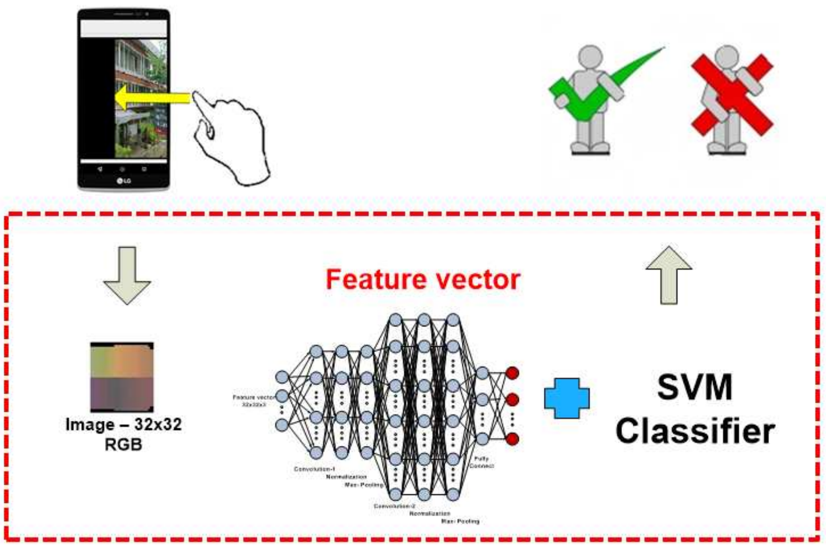


Figure 4.3 – User authentication process with the architecture trained in Cifar dataset

We analyzed the output of the first fully connected layer (Output of 128) and the output of the second convolutional layer (Output of 6400), with an SVM at the end of the architecture, as we can see in the Figure 4.3. We used the Cifar architecture trained in Cifar dataset architecture as a feature extractor of images RGB, feeding with that the SVM classifier (generating a model for user authentication). Besides, we used fine-tuning with our images RGB based on gestures to update the weights in the architecture of Cifar dataset to increase the accuracy.

The new model of user authentication was implemented on Android using Torch Android [3]. Besides, we incorporated two functions in our application to collect data and generate the model for user authentication into the mobile environment data and generate the model for user authentication into the mobile environment.

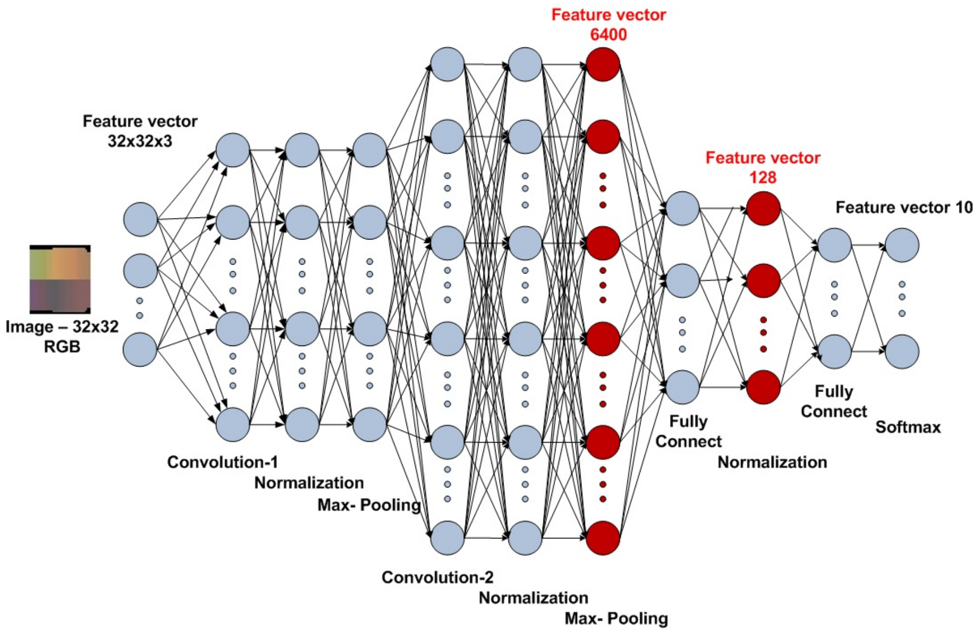


Figure 4.4 – Transfer learning scheme with Cifar-10 Architecture

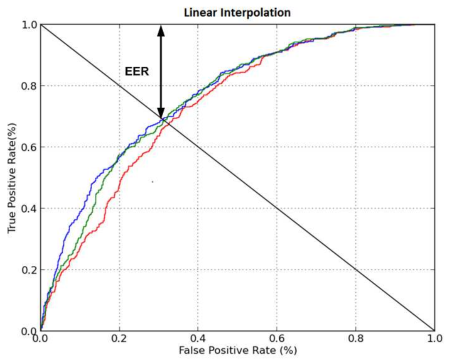
4.2.3 GTGF-RGB Results

The EER is the False Positive Rate or the False Negative Rate at the point on the Operating Curve where those numbers are equal. It can be visualized at the intersection between the Receiver Operating Characteristic (ROC) curve and the black straight line connecting [100%, 0%] (Figure 4.5).

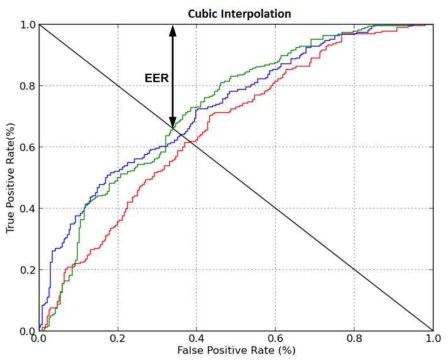
(ZHAO; FENG; SHI, 2013) reports an EER between 6.33 and 15.3, which we could not reproduce: as seen in Figure 4.5b, our best results in this approach are EER - 36.8 using cubic interpolation and EER - 35.5 using linear interpolation.

Table 4.2 shows the results with GTGF-RGB, the new technique we proposed in Section 4.2.2, where the feature vector in Cifar-1 and Cifar-2 is 128 and 6400, respectively.

[3] https://github.com/soumith/torch-android



(a) Results of image with linear interpolation



(b) Results of image with cubic interpolation

Figure 4.5 – ROC curves of metrics scores (Red-NCC, Green-L1, Blue-L2)

The term TL is referred to the use of transfer learning after applying fine-tuning. The best results were obtained using a Cifar-2-TL with SVM-RBF classifier in the last layer in Touchalytics and RECOD–Gesture datasets with 17.0% and 10.3 % of EER, respectively.

Table 4.2 – Gestures EER results in Deep learning

Touchalytics	**SVM - Linear**	**SVM - RBF**	**Final Feature Size**
Cifar-1	17.3	15.6	128
Cifar-2	19.6	14.0	6400
Cifar-1 - TL	22.0	13.2	128
Cifar-2 - TL	17.0	13.0	6400
RECOD–Gestures			
Cifar-1	14.0	13.2	128
Cifar-2	13.0	12.6	6400
Cifar-1 - TL	13.8	12.5	128
Cifar-2 - TL	10.3	10.2	6400

The system is robust to detect different types of gestures with a high level of certainty, around EER-10 in normal conditions (Figure 4.6). However, in different scenarios, our proposed model can not be used for all mobile applications because the user interaction with the mobile changes a lot among the applications.

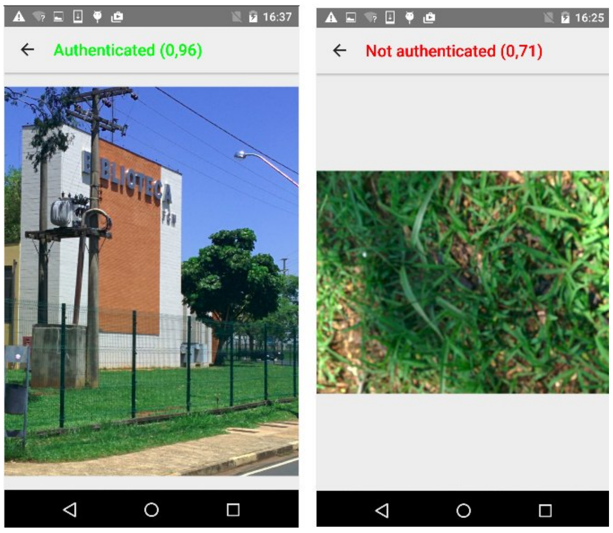


Figure 4.6 – User Authentication based in gestures

4.3 Second Solution: Time Series and Recurrent Networks

Our second implementation forgoes the encoding of the gestures into images and processes the time series directly using LSTMs. In order to fit the architecture in hardware (using the Mini Mind design described in the previous chapter), we used a relatively shallow (for today's standards) architecture, with two LSTM layers and two fully connected layers.

We started with a reference implementation in software, which we used as a baseline for the hardware solution. We expected the software implementation to have slightly better accuracy because of numerical approximation: the network in software uses 32-bit floating-point numbers, while Mini Mind's hardware uses 16-bit fixed-point numbers; the network in software uses high-precision approximations for the transcendental functions, while Mini Mind's hardware uses very rough approximations (as explains in Section 3.2.2). So the aim here is not for the hardware implementation to *surpass* the software baseline: that would be unrealistic. The aim is for the hardware implementation to be as close as possible to the software baseline.

4.3.1 Reference implementation in software

For the design and implementation of any system in hardware, it is necessary to know all the values of each unit. This value (Floating point of precision) is used as a reference as the maximum value or the ideal value to get around the implementation in hardware. When the precision is changed (Fixed point of precision), it is usual in these cases to have similar results with a slight variation in decimals.

In our case, for all training processes, We used Keras[4] as the framework for making a recurrent neural network model.

Mini Mind model has two layers of LSTM, two layers of fully connected, and one softmax layer to authenticate each user (In our case, we have a multiclass problem with 41 users because of a dataset). After gesture normalization, it was necessary to divide all gesture datasets into train, validation, and test datasets. For this purpose, we selected in a random way per user 50% for training, 25% for validation, and the rest for the test.

We used root mean square propagation (RMSProp) as an optimizer and cross-entropy as a cost function. We used a GPU Tesla k40 to train the model during 1000 epochs, in which the best epoch was in 302 with an accuracy of 72%. We selected the layers of LSTM and fully connected them because they were the most expensive part of the processing, which required 233.5 operations in each layer.

4.3.2 Results Software vs. Hardware

This section will use the results obtained using python/Keras, which will be used in the hardware implementation process.

One of the essential factors in the implementation is the synchronization of the blocks. The synchronization with the basics elements in a complex module plays an important role. Usually, this process is significantly delayed, and it is recommended to do that in parts. During the creation of the Mini Min's units, we guaranteed that each module worked correctly, but when these modules are interconnected in the intricate design - combining units into the LSTM layer - delays should be considered, not only at the level of basic units -logic gates- but also at the level of LSTM layer.

Figure 4.7 shows the error rate hardware and software in LSTM layer. To achieve these results, We compare the outputs of the units *input gate* (2.2), *forget gate* (2.3), *output gate* (2.4), corresponding in the graph as unit-1, unit-2 and unit-3. And also, the hardware response, using 8-bit to represent integers and also 8 bits for the fractional part. We can conclude that the use of precision for this type of units is not a critical factor for its implementation in hardware. In addition, it was used a piecewise linear approximation for nonlinear functions such as sigmoid and tanh described in section 3.2.2.

In Figure 4.8, we can see that the error rate grows progressively at each time iteration. This is a consequence when we add another layer, where the label Python result is the response of the system using floating-point precision, and we consider that value as a reference. Python mod Tanh / Sig label refers that the response is not using the functions provided by math operation in python, but it uses a piecewise approximation, which limits his precision, and we can have it is closer to the one implemented in hardware. FPGA

[4] Keras is a high-level neural networks API, written in Python https://keras.io/

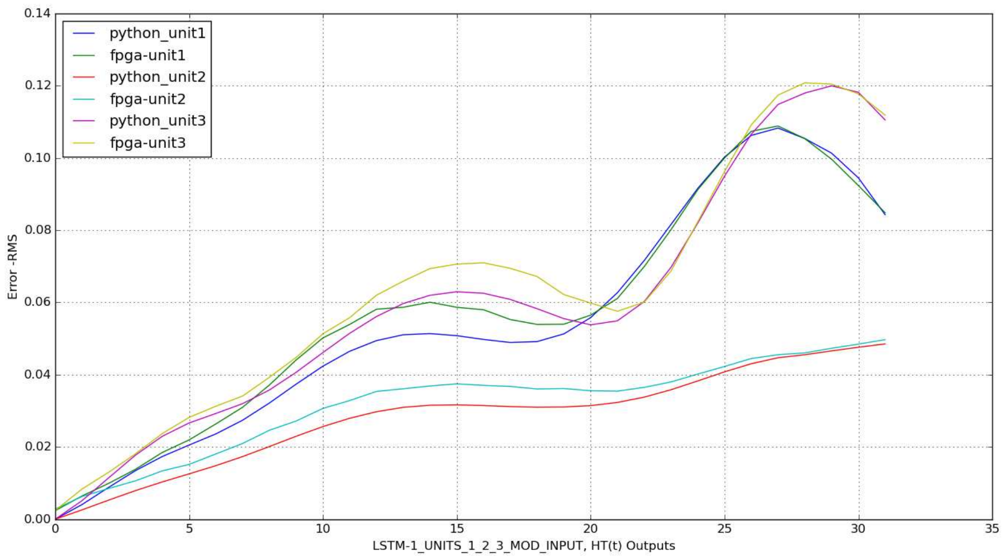


Figure 4.7 – Result of one LSTM layer Hardware vs Software

- ModelSim label refers to the result obtained in the hardware simulator (Testbench). Moreover, FPGA - Hardware is the expected result already in hardware-implemented

As we expected, ModelSim and FPGA Hardware results are the same. However, when we compare it with the other results, we can see that the error increases exponentially when we add another LSTM layer.

The first idea that we had to solve this problem was to change the precision, and we thought that the precision was not enough because of the complexity of the model. However, this idea was discarded after analyzing the previous state in Figure 4.7, where we can see the representation and unit's result and conclude that it is possible to represent that level with that type of precision.

Another idea that we were the approximation of the non-linear functions, as we know it is not possible to use non-linear functions in an FPGA. This only supports linear systems, so it is necessary to approximate the next value of the real function. For our design, using piecewise linear approximation level 4 to sigmoid and 12 for the tangent function (Calculating in section 3.2.2) does not help to improve the final result due to the precision of 16 bits that the system has.

After exhaustively analyzing all the components, it was possible to conclude that when we include another layer of LSTM, we also need to consider: the initial delay, the delay through all components until it reaches the second unit (Among LSTM layers), and finally the delay to init all blocks and basic units at the same time. This new approach had not been considered in the original design. When we include the second layer, all the basics units did have the initial delay value to synchronize with the first layer, and the

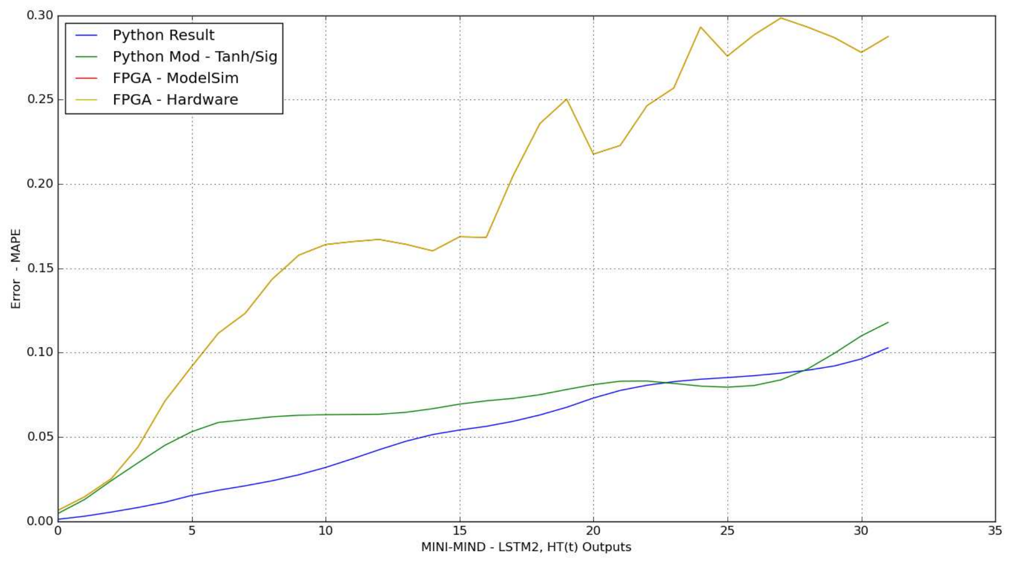


Figure 4.8 – Result the second LSTM layer Hardware vs Software

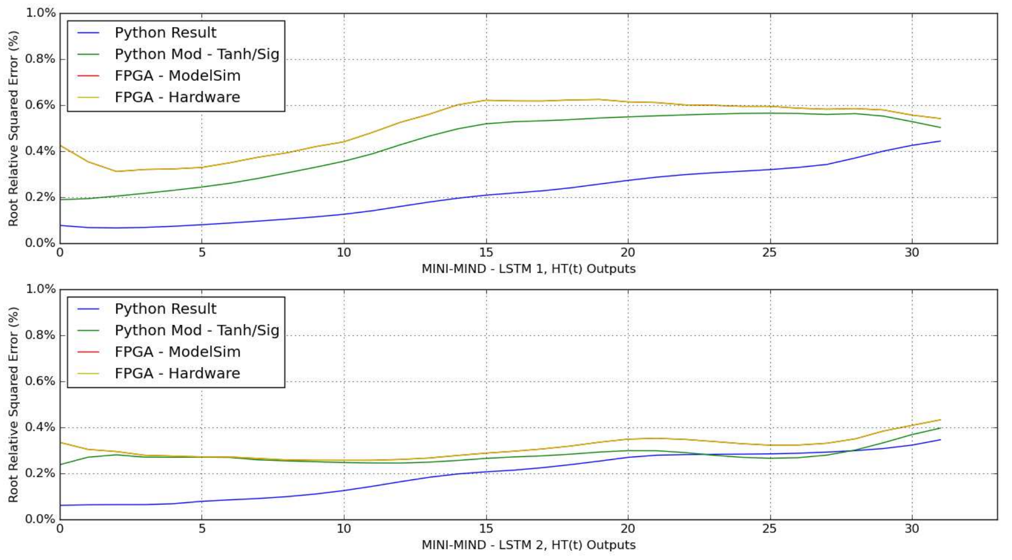


Figure 4.9 – Result of two LSTM layers Hardware vs Software

error grows exponentially by a lack of synchronization among the different layers.

In Figure 4.9, we can see that this type of lack of synchronization among layers was solved by just increasing another timer trigger into the second layer. If we need to increase the number of layers, we also need to increase the value of time in this trigger (Depending on the number of neurons in each layer).

In Figure 4.10,we can see final result of two layers LSTM. As we expected at the beginning, the results in the software are the same as hardware (simulated) and in hardware implemented with the software value.

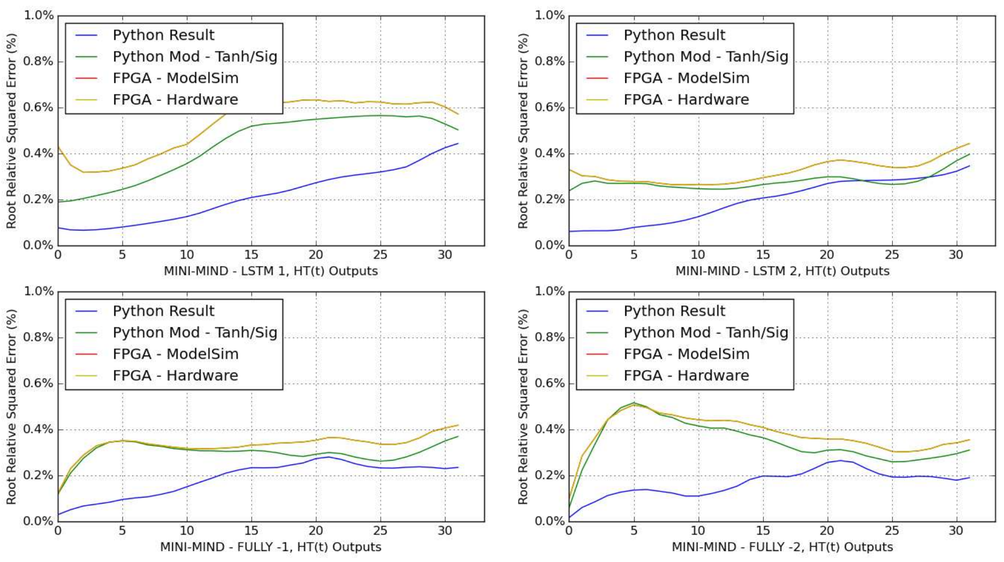


Figure 4.10 – Final Result of two LSTM layers Hardware vs Software

Table 4.3 – Comparison about hardware architectures

Hardware	**Time (s)**	**Performance (W)**	**Mop/s/W**
Tesla K40 – GPU	0.193	55.00	22.01
Lenovo G400s – CPU	0.035	8.81	757.17
Cyclon IV – FPGA	0.014	0.50	34347.41

4.3.3 Results and discussion

The accuracies in the test split of the dataset 4.1.0.2 (with 4476 gestures) are:

- Using the baseline software implementation in Python/Keras: 71.16%;
- Using an algorithm in python (CPU) with piecewise tanh approximation - using the same algorithm as keras but without the use of non-linear functions, just linear approximations made in python-, we obtain 69.30% of accuracy. That means using python
- Using the hardware implementation in Mini Mind: 69.30%.

The software's accuracy (Tanh approximation) and hardware are the same because 16 bits of precision in Mini Mind and the floating-point operation in CPU give the same results. However, if we compare the result provided by Keras (Full tanh function without approximations), we have a 1.86% of loss in the test dataset because of tanh approximation function.

Table 4.3 shows our results for each platform when they are processing one single gesture because a continuous authentication approach must evaluate one gesture

Table 4.4 – FPGA Hardware resource utilization for Cyclone IV

Component	**Utilization**[/]	**Utilization**[%]
Logic Elements Combinational	5563	4
Logic Elements Registers	3808	4
Total Memory Bits	3747840	56
Embedded Multiplier 9-bit Element	68	9

per time. Mini Mind hardware can compute one gesture 2.5 times faster than CPU and 14 times faster than GPU. In terms of power consumption, it can consume 17 times less than CPU and 109 times less than GPU. Moreover, Table 4.4 shows our results when we implement a Mini Mind in Cyclone IV, in which case, embedded memory is the most component used in our design because of the use of small memories around all design.

5 Conclusion

This work presented a novel architecture for deep learning, Mini Mind, whose main contributions are:

- The implementation of all LSTM layers into basic units.
- Optimizing the matrix-vector multiplication using MAC, adders, and registers in a parallel form.
- The use of small and temporal memories allows Mini Mind the possibility to deploy in a pipeline form.
- Low Power consumption, 17 times less than a CPU and 109 times less than a GPU.
- Bus data interconnection with Nios II processor by Avalon for testing.

Mini Mind could obtain results comparable to software running in CPUs or GPUs, obtaining a high throughput in FPGA hardware. Mini Mind contains two layers of LSTM and fully connects with 128 units in hardware.

The MAC units' use was already optimized for space, not for speed. In addition, if we increase the numbers of elements (Basic units and MACs elements) in a pipeline process, the final throughput would increase.

The focus of this work was to obtain the best result possible using a pipeline architecture. For this reason, the use of memories for weights was not optimized. However, we could change the precision from floating-point to fixed-point (8.8), reducing the number of elements in hardware considerably. Finally, small and temporal memories allow the Mini Mind to use a pipeline form, obtaining a high throughput in FPGA hardware.

One of the main challenges regarding Mini Mind design was the interconnection of the units. Numerical or synchronization errors in this process lead to errors that grow exponentially as the processing cycles accumulate due to the recurrent nature of LSTMs. Thus, careful interconnection must be done among all basic units and layers. The first type of interconnection error, among basics units, was related to a delay caused by all the units within an LSTM layer, and it was not considered correctly because it was only considered for each unit. The second type of interconnection error, among modules and layers, was solved, including an additional control unit to each layer, to guarantee all synchronous processes from beginning to end of the LSTM layer.

Moreover, the interaction with a generic processor (NIOS II) helped us to emulate an actual situation where the MINI MIND requires incorporating the weights for

LSTM layers and demands additional elements to implement the design, showcasing the speed of the pipeline design.

We validated our hardware design in the application of continuous user authentication using gesture recognition via three platforms:

- A software Keras we have a 71.16% of accuracy.
- An algorithms in python in a CPU with piecewise tanh approximation we obtained 69.30% of accuracy
- And finally, in Mini Mind hardware accuracy was 69.30% of accuracy

In summary, the software's accuracy (With TanH approximation) and hardware are the same because of the use of 16 bits of precision in hardware. However, if we compare the result provided by Keras, we have a small (1.86%) loss in the test dataset because of that approximation. Mini Mind hardware can process one gesture 2.5 times faster than a CPU and 14 times faster than a GPU.

In addition to the contributions above, we have also tested an additional *software solution for gesture recognition*, encoding gestures into RGB images. That method was implemented on Android devices, using a deep convolutional network pre-trained on CIFAR-10 as a feature extractor. The result was optimal for normal conditions where the user interacts reasonably in a single application. Across different applications, however, the system was not sufficiently robust to identify the users. That limitation: consistency of behavior across tasks/applications may be the most severe hindrance for the large scale of gesture recognition for continuous authentication. Another limitation we faced was the available sensors, for example, a touchscreen incapable of measuring pressure values directly.

More interestingly, our proposed method had a large memory footprint, producing some delay problems during the authentication process and excessive consumption of batteries, which would limit their applications in a natural environment. We believe those limitations may stimulate the industry to incorporate dedicated accelerators like Mini Mind for specific Machine Learning tasks that must run continuously in consumer electronics without impacting other applications.

5.1 Next Steps

We identify four research fronts as exciting avenues for future works.

First, **binary networks**. One of the areas that we would most like to explore for future work is binary networks, which use fast logical operations (like XORs) instead of expensive arithmetics (like multiplications). Those types of networks can guarantee full parallelism in any FPGA platform. Their major disadvantage is a major loss in accuracy (10% points or more). It would be an exciting research discovery for hardware implementation if those architectures could be more accurate.

Second, **quantization and pruning**. Currently, those methods are most used for hardware implementation, combining proper management of resources, which proposes many approaches to reduce the accuracy and weights of the network, which was little explored in this work.

Third, the use of **dynamic precision**. In the last year, manufacturers of onboard devices such as Xilinx and Altera used dynamic precision to represent complex networks, reducing precision to as low as 8 bits in complex neural networks. Those approaches have synergy with quantization and pruning mentioned above. In combination, both are a natural next step in future works.

And finally, the application of**continuous authentication**. We believe that adopting dedicated hardware accelerators will be unavoidable for deploying continuous authentication to mobile devices. Each solution, in hardware or software, presents complementary advantages and disadvantages. Thus the combination of both could be an exciting pursuit. For example, a simple solution in hardware could run all the time to eliminate most of the intruders, while a more sophisticated solution in software could run from time to time to eliminate the rest. By combining both hardware and software and considering several factors beyond gesture recognition, we believe continuous authentication could reach very high accuracy.

Bibliography

2014 eMarketer. *2 Billion Consumers Worldwide to Get Smart(phones) by 2016*. Cited in page 14.

2019 statista. *Smartphone users worldwide 2016-2021*. Cited in page 14.

ABDELOUAHAB, K. et al. Accelerating cnn inference on fpgas: A survey. *arXiv preprint arXiv:1806.01683*, 2018. Cited 2 in pages 24 and 32.

ALTERA. In: ALTERA CORPORATION. *Creating Multiprocessor Nios II Systems Tutorial*. 2011. Disponível em: <https://www.altera.com/en_US/pdfs/literature/tt/tt_nios2_multiprocessor_tutorial.pdf>. Cited in page 50.

ALTERA. In: ALTERA CORPORATION. *Internal Memory (RAM and ROM) User Guide*. 2014. Disponível em: <https//www.altera.com/content/dam/altera-www/global/en_US/pdfs/literature/ug/ug_ram.pdf>. Cited in page 46.

AVIV, A. J. et al. Smudge attacks on smartphone touch screens. *Woot*, v. 10, p. 1–7, 2010. Cited in page 15.

BARKALOV, A.; TITARENKO, L.; MAZURKIEWICZ, M. *Foundations of Embedded Systems*. [S.l.]: Springer, 2019. v. 195. Cited in page 38.

BENGIO, I. G. Y.; COURVILLE, A. *Deep Learning*. [s.n.], 2016. Book in preparation for MIT Press. Disponível em: <http://www.deeplearningbook.org>. Cited 2 in pages 14 and 19.

BENGIO, Y.; SIMARD, P.; FRASCONI, P. Learning long-term dependencies with gradient descent is difficult. *Neural Networks, IEEE Transactions on*, IEEE, v. 5, n. 2, p. 157–166, 1994. Cited in page 21.

CADAMBI, S. et al. A programmable parallel accelerator for learning and classification. In: ACM. *Proceedings of the 19th international conference on Parallel architectures and compilation techniques*. [S.l.], 2010. p. 273–284. Cited in page 30.

CAO, S. et al. Efficient and effective sparse lstm on fpga with bank-balanced sparsity. In: ACM. *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. [S.l.], 2019. p. 63–72. Cited in page 33.

CARVALHO, M. *Transfer schemes for deep learning in image classification*. Dissertação (Mestrado) — Universidade Estadual de Campinas . Faculdade de Engenharia Elétrica e de Computação, 2015. Cited in page 59.

CARVALHO, M. et al. Deep neural networks under stress. *arXiv preprint arXiv:1605.03498*, 2016. Cited in page 27.

CEVA. *CEVA Deep Neural Network (CDNN)*. Disponível em: <http://www.ceva-dsp.com/CDNN2>. Cited 2 in pages 9 and 17.

CHAKRADHAR, S. et al. A dynamically configurable coprocessor for convolutional neural networks. In: ACM. *ACM SIGARCH Computer Architecture News*. [S.l.], 2010. v. 38, n. 3, p. 247–257. Cited in page 30.

CHANG, A. X. M.; CULURCIELLO, E. Hardware accelerators for recurrent neural networks on fpga. In: *2017 IEEE International Symposium on Circuits and Systems (ISCAS)*. [S.l.: s.n.], 2017. p. 1–4. Cited 4 in pages 9, 30, 32, and 41.

CLOUTIER, J. et al. Vip: An fpga-based processor for image processing and neural networks. In: IEEE. *microneuro*. [S.l.], 1996. p. 330. Cited 2 in pages 9 and 25.

COURBARIAUX, M.; BENGIO, Y.; DAVID, J.-P. Training deep neural networks with low precision multiplications. *arXiv preprint arXiv:1412.7024*, 2014. Cited 2 in pages 9 and 27.

COURBARIAUX, M.; BENGIO, Y.; DAVID, J.-P. Binaryconnect: Training deep neural networks with binary weights during propagations. In: *Advances in Neural Information Processing Systems*. [S.l.: s.n.], 2015. p. 3105–3113. Cited in page 29.

COX, C. E.; BLANZ, W. E. Ganglion-a fast field-programmable gate array implementation of a connectionist classifier. *Solid-State Circuits, IEEE Journal of*, IEEE, v. 27, n. 3, p. 288–299, 1992. Cited 2 in pages 9 and 24.

DUNDAR, A. et al. Embedded streaming deep neural networks accelerator with applications. IEEE, 2016. Cited 2 in pages 29 and 30.

EKAS, P.; JENTZ, B. Developing and integrating fpga coprocessors. *Embedded Computing Design Magazine*, 2003. Cited in page 50.

ELMAN, J. L. Finding structure in time. *Cognitive science*, Wiley Online Library, v. 14, n. 2, p. 179–211, 1990. Cited in page 21.

FARABET, C. et al. Cnp: An fpga-based processor for convolutional networks. p. 32–37, 2009. Cited 3 in pages 25, 26, and 30.

FENG, T. et al. Continuous mobile authentication using touchscreen gestures. In: IEEE. *Homeland Security (HST), 2012 IEEE Conference on Technologies for*. [S.l.], 2012. p. 451–456. Cited 2 in pages 35 and 36.

FENG, T. et al. Tips: Context-aware implicit user identification using touch screen in uncontrolled environments. In: ACM. *Proceedings of the 15th Workshop on Mobile Computing Systems and Applications*. [S.l.], 2014. p. 9. Cited 2 in pages 36 and 37.

FRANK, M. et al. Touchalytics: On the applicability of touchscreen input as a behavioral biometric for continuous authentication. *Information Forensics and Security, IEEE Transactions on*, IEEE, v. 8, n. 1, p. 136–148, 2013. Cited 5 in pages 36, 37, 54, 56, and 57.

FUKUSHIMA, K. Cognitron: A self-organizing multilayered neural network. *Biological cybernetics*, Springer, v. 20, n. 3-4, p. 121–136, 1975. Cited in page 20.

FUKUSHIMA., K. Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position. *Biological Cybernetics*, Springer, v. 36, n. 4, p. 193–202, 1980. Cited in page 20.

GOKHALE, V. et al. A 240 g-ops/s mobile coprocessor for deep neural networks. In: *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR) Workshops*. [S.l.: s.n.], 2014. Cited 3 in pages 9, 26, and 27.

GOODFELLOW, I. J. et al. Maxout networks. *arXiv preprint arXiv:1302.4389*, 2013. Cited in page 27.

GREFF, K. et al. Lstm: A search space odyssey. *arXiv preprint arXiv:1503.04069*, 2015. Cited in page 30.

GUO, K. et al. Angel-eye: A complete design flow for mapping cnn onto embedded fpga. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, IEEE, v. 37, n. 1, p. 35–47, 2018. Cited in page 33.

GUPTA, S. et al. Deep learning with limited numerical precision. *arXiv preprint arXiv:1502.02551*, 2015. Cited in page 27.

HAN, S. et al. Ese: Efficient speech recognition engine with sparse lstm on fpga. In: *FPGA*. [S.l.: s.n.], 2017. p. 75–84. Cited in page 31.

HAN, S. et al. Eie: efficient inference engine on compressed deep neural network. In: IEEE PRESS. *Proceedings of the 43rd International Symposium on Computer Architecture*. [S.l.], 2016. p. 243–254. Cited in page 30.

HAN, S.; MAO, H.; DALLY, W. J. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. *arXiv preprint arXiv:1510.00149*, 2015. Cited in page 29.

HE, K. et al. Deep residual learning for image recognition. In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. [S.l.: s.n.], 2016. p. 770–778. Cited in page 22.

HEBB, D. O. *The Organization of Behavior: A Neuropsychological Theory*. New York: Wiley, 1949. Hardcover. ISBN 0805843000. Cited in page 19.

HILL, K. et al. Comparative analysis of opencl vs. hdl with image-processing kernels on stratix-v fpga. In: IEEE. *Application-specific Systems, Architectures and Processors (ASAP), 2015 IEEE 26th International Conference on*. [S.l.], 2015. p. 189–193. Cited in page 32.

HINTON, G. E.; OSINDERO, S.; TEH, Y.-W. A fast learning algorithm for deep belief nets. *Neural computation*, MIT Press, v. 18, n. 7, p. 1527–1554, 2006. Cited in page 22.

HINTON, G. E. et al. Improving neural networks by preventing co-adaptation of feature detectors. *arXiv preprint arXiv:1207.0580*, 2012. Cited in page 22.

HOCHREITER, S.; SCHMIDHUBER, J. Long short-term memory. *Neural computation*, MIT Press, v. 9, n. 8, p. 1735–1780, 1997. Cited in page 22.

HOPFIELD, J. J. Neural networks and physical systems with emergent collective computational abilities. *Proceedings of the national academy of sciences*, National Acad Sciences, v. 79, n. 8, p. 2554–2558, 1982. Cited in page 20.

INTEL. In: INTEL CORPORATION. *Embedded Design Handbook*. 2016. Disponível em: <https://www.altera.com/en_US/pdfs/literature/hb/nios2/edh_ed_handbook.pdf>. Cited in page 50.

JIANG, W. et al. Accuracy vs. efficiency: Achieving both through fpga-implementation aware neural architecture search. *arXiv preprint arXiv:1901.11211*, 2019. Cited in page 33.

JORDAN, M. Serial oprder: A parallel distributed processing approach (tech. rep. no. 8604). *San Diego, La Jolla, CA: Institute for Cognitive Science, University of California. Cited on*, p. 48, 1986. Cited in page 21.

KIEFER, J.; WOLFOWITZ, J. et al. Stochastic estimation of the maximum of a regression function. *The Annals of Mathematical Statistics*, Institute of Mathematical Statistics, v. 23, n. 3, p. 462–466, 1952. Cited in page 20.

KRIZHEVSKY, A.; SUTSKEVER, I.; HINTON, G. E. Imagenet classification with deep convolutional neural networks. p. 1097–1105, 2012. Cited in page 22.

LECUN, Y.; BENGIO, Y. Convolutional networks for images, speech, and time series. *The handbook of brain theory and neural networks*, v. 3361, n. 10, p. 1995, 1995. Cited in page 21.

LECUN, Y. et al. Lenet-5, convolutional neural networks. *URL: http://yann. lecun. com/exdb/lenet*, v. 20, p. 5, 2015. Cited in page 26.

LI, L. et al. An integrated hardware/software design methodology for signal processing systems. *Journal of Systems Architecture*, Elsevier, v. 93, p. 1–19, 2019. Cited in page 38.

LI, N. et al. A multistage dataflow implementation of a deep convolutional neural network based on fpga for high-speed object recognition. In: IEEE. *2016 IEEE Southwest Symposium on Image Analysis and Interpretation (SSIAI)*. [S.l.], 2016. p. 165–168. Cited 2 in pages 29 and 30.

LIN, M.; CHEN, Q.; YAN, S. Network in network. *arXiv preprint arXiv:1312.4400*, 2013. Cited in page 29.

LUCA, A. D. et al. Touch me once and i know it's you!: implicit authentication based on touch screen patterns. In: ACM. *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. [S.l.], 2012. p. 987–996. Cited 2 in pages 35 and 36.

MCCULLOCH, W. S.; PITTS, W. A logical calculus of the ideas immanent in nervous activity. *The bulletin of mathematical biophysics*, Springer, v. 5, n. 4, p. 115–133, 1943. Cited in page 19.

MINSKY, M.; PAPERT, S. Perceptron: an introduction to computational geometry. *The MIT Press, Cambridge, expanded edition*, v. 19, n. 88, p. 2, 1969. Cited in page 20.

MISRA, J.; SAHA, I. Artificial neural networks in hardware: A survey of two decades of progress. *Neurocomputing*, Elsevier, v. 74, n. 1-3, p. 239–255, 2010. Cited 2 in pages 23 and 24.

NGUYEN, T. V.; SAE-BAE, N.; MEMON, N. Draw-a-pin: Authentication using finger-drawn pin on touch devices. *Computers & Security*, Elsevier, v. 66, p. 115–128, 2017. Cited 2 in pages 36 and 37.

OMONDI, A. R.; RAJAPAKSE, J. C. *FPGA implementations of neural networks*. [S.l.]: Springer, 2006. v. 365. Cited in page 21.

PAN, S. J.; YANG, Q. A survey on transfer learning. *Knowledge and Data Engineering, IEEE Transactions on*, IEEE, v. 22, n. 10, p. 1345–1359, 2010. Cited 2 in pages 22 and 59.

PATEL, V. M. et al. Continuous user authentication on mobile devices: Recent progress and remaining challenges. *IEEE Signal Processing Magazine*, IEEE, v. 33, n. 4, p. 49–61, 2016. Cited 3 in pages 15, 34, and 35.

PEEMEN, M. et al. Memory-centric accelerator design for convolutional neural networks. In: IEEE. *Computer Design (ICCD), 2013 IEEE 31st International Conference on*. [S.l.], 2013. p. 13–19. Cited in page 30.

POPULATIONPYRAMID. *Population Pyramids of the World from 1950 to 2100*. Cited in page 14.

RAHMAN, A.; LEE, J.; CHOI, K. Efficient fpga acceleration of convolutional neural networks using logical-3d compute array. In: IEEE. *2016 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. [S.l.], 2016. p. 1393–1398. Cited 2 in pages 28 and 30.

RASTEGARI, M. et al. Xnor-net: Imagenet classification using binary convolutional neural networks. *arXiv preprint arXiv:1603.05279*, 2016. Cited 2 in pages 22 and 29.

ROBBINS, H.; MONRO, S. A stochastic approximation method. *The annals of mathematical statistics*, JSTOR, p. 400–407, 1951. Cited in page 20.

ROSENBLATT, F. The perceptron: a probabilistic model for information storage and organization in the brain. *Psychological review*, American Psychological Association, v. 65, n. 6, p. 386, 1958. Cited in page 19.

ROY, A.; HALEVI, T.; MEMON, N. An hmm-based behavior modeling approach for continuous mobile authentication. In: IEEE. *Acoustics, Speech and Signal Processing (ICASSP), 2014 IEEE International Conference on*. [S.l.], 2014. p. 3789–3793. Cited in page 36.

RUMELHART, D. E.; HINTON, G. E.; WILLIAMS, R. J. *Learning internal representations by error propagation*. [S.l.], 1985. Cited in page 20.

RUMELHART, D. E. et al. *Parallel distributed processing*. [S.l.]: IEEE, 1988. v. 1. Cited in page 20.

SAE-BAE, N.; MEMON, N.; ISBISTER, K. Investigating multi-touch gestures as a novel biometric modality. In: IEEE. *Biometrics: Theory, Applications and Systems (BTAS), 2012 IEEE Fifth International Conference on*. [S.l.], 2012. p. 156–161. Cited 2 in pages 35 and 36.

SANKARADAS, M. et al. A massively parallel coprocessor for convolutional neural networks. p. 53–60, 2009. Cited in page 30.

SHAWAHNA, A.; SAIT, S. M.; EL-MALEH, A. Fpga-based accelerators of deep learning networks for learning and classification: A review. *IEEE Access*, IEEE, v. 7, p. 7823–7859, 2019. Cited in page 24.

SUDA, N. et al. Throughput-optimized opencl-based fpga accelerator for large-scale convolutional neural networks. p. 16–25, 2016. Cited 5 in pages 9, 11, 23, 29, and 30.

SZE, V. et al. Hardware for machine learning: Challenges and opportunities. *arXiv preprint arXiv:1612.07625*, 2016. Cited in page 15.

SZE, V. et al. Efficient processing of deep neural networks: A tutorial and survey. *arXiv preprint arXiv:1703.09039*, 2017. Cited 3 in pages 11, 22, and 23.

SZEGEDY, C. et al. Going deeper with convolutions. In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. [S.l.: s.n.], 2015. p. 1–9. Cited in page 22.

VIDAL, M.; CRUCES, R.; ZURITA, G. Digital fir filter design for diagnosing problems in gears and bearings using xilinx's system generator. *moment*, v. 1, p. 0–071, 2014. Cited in page 38.

WANG, C. et al. Dlau: A scalable deep learning accelerator unit on fpga. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, IEEE, v. 36, n. 3, p. 513–517, 2017. Cited in page 30.

WANG, Z. et al. Various frameworks and libraries of machine learning and deep learning: A survey. *Archives of Computational Methods in Engineering*, Springer, p. 1–24, 2019. Cited in page 33.

WERBOS, P. J. *Beyond Regression: New Tools for Prediction and Analysis in the Behavioral Sciences*. Tese (Doutorado) — Harvard University, 1974. Department of Applied Mathematics. Cited in page 20.

WERBOS., P. J. Backpropagation through time: what it does and how to do it. *Proceedings of the IEEE*, IEEE, v. 78, n. 10, p. 1550–1560, 1990. Cited in page 20.

WIDROW, B. Pattern recognition and adaptive control. *Applications and Industry, IEEE Transactions on*, IEEE, v. 83, n. 74, p. 269–277, 1964. Cited in page 20.

WIDROW, B.; HOFF, M. E. Associative storage and retrieval of digital information in networks of adaptive "neurons". In: *Biological Prototypes and Synthetic Systems*. [S.l.]: Springer, 1962. p. 160–160. Cited in page 20.

ZHANG, C. et al. Optimizing fpga-based accelerator design for deep convolutional neural networks. p. 161–170, 2015. Cited 4 in pages 9, 19, 28, and 30.

ZHAO, X.; FENG, T.; SHI, W. Continuous mobile authentication using a novel graphic touch gesture feature. In: IEEE. *Biometrics: Theory, Applications and Systems (BTAS), 2013 IEEE Sixth International Conference on*. [S.l.], 2013. p. 1–6. Cited 6 in pages 36, 37, 54, 56, 57, and 60.