UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM MICROELETRÔNICA

LEANDRO MATEUS GIACOMINI ROCHA

Energy-Efficient Recurrent Neural Network Hardware Architecture for Heart Rate Estimation Based on Photoplethysmography

Advisor: Prof. Dr. Sergio Bampi

Porto Alegre
November 2020

CIP - Catalogação na Publicação

Giacomini Rocha, Leandro Mateus
Energy-Efficient Recurrent Neural Network Hardware Architecture for Heart Rate Estimation Based on Photoplethysmography / Leandro Mateus Giacomini Rocha. -- 2020.
159 f.
Orientador: Sergio Bampi.

Tese (Doutorado) -- Universidade Federal do Rio Grande do Sul, Instituto de Informática, Programa de Pós-Graduação em Microeletrônica, Porto Alegre, BR-RS, 2020.

1. Neural Networks. 2. VLSI implementation. 3. Heart Rate estimation. 4. Photoplethysmography. 5. Low Power. I. Bampi, Sergio, orient. II. Título.

Elaborada pelo Sistema de Geração Automática de Ficha Catalográfica da UFRGS com os dados fornecidos pelo(a) autor(a).

“If I have seen farther than others,
it is because I stood on the shoulders of giants.”
— Sir Isaac Newton

ACKNOWLEDGEMENTS

First and foremost, I would like to thank my tutor Prof. Sergio Bampi. He has believed in me since I was in my last graduation semester. Professor Sergio has been an inspiration to me for all his achievements and his will to help us rise and shine in our personal and professional lives. I appreciate his patience and guidance during the entire development of this thesis.

Second, this Ph.D. would never be possible would the permanent support of my dear Maiquidieli Dal Berto, the love of my life, my best friend, and the best companion I could ever have. As a Ph.D. herself, she always understood the emotional roller coaster that is a Ph.D. thesis with all the anxiety, frustration, and celebration inherent to this process. Her unwavering support helped me go through tough days, especially during my internship in Belgium when we were apart. I could never find a more loving, caring, and perfect person who makes me be a better person every day, and I will always love to the moon and back.

During my Ph.D., I made a lot of friends, which I profoundly care about whose friendship I will always foster for the rest of my life. Particularly, I must thank Guilherme Paim and Eduardo da Costa, two wonderful people who are constantly seeking more research areas that can explore. Their help was quintessential during this thesis, especially with all the late nights and the never-ending barbecues.

I cannot express how grateful I am for all my colleagues in Laboratory 215, which always believed in the teamwork principle. They were always there for fruitful discussions and sharing a cup of coffee. In particular, I would like to thank Brunno Abreu, Gustavo Santana, Thomas Fontanari, Ana Mativi, Mateus Grellert, Dieison Silveira, Leonardo Soares, Eduarda Monteiro. For sure, you create a research environment that encourages students to explore the challenges of academic research.

I must also thank my best friends João de Carli and Fernanda Gaboardi, whose friendship is unpaired. The bond created in all these years is something that really makes me happy. Sharing fears, thoughts, and joys were essential to keep sanity in this crazy world.

I must also thank all those people who helped in every way during my stay in Leuven. Particularly, I would like to thank Dwaipayan Biswas for his support during my internship at IMEC. He not only helped me to adapt to the city but also shared insightful ideas for developing this work. I would also like to thank Prof. Marian Verhelst for all

the fruitful meetings and her help with university matters.

This work was only possible because my family has always supported me, providing love, affection, and everything else I ever needed. Special thanks to my father Ivan Rocha and my mother, Marilise Giacomini Rocha, for always fostering in me the desire for big dreams.

Finally, thank you CAPES, CNPq, IFRS, and every funding agency that supported my studies. I will put all my efforts into proving myself worthy of all your investments, aiming to improve and expand the Brazilian scientific community.

ABSTRACT

The increasing power density and the pervasive use of compute-intensive and power-hungry applications demand energy-efficient CMOS design. The quest for energy-efficient systems particularly concerns in wearable devices for health monitoring as they must be under non-stop operation with limited energy source available on miniaturized batteries. There is an ever-growing interest in employing neural network-based applications for data processing on edge devices. Neural networks have complex structures in their pure software or hardware implementations, or in a combination of both approaches. They require millions of data fetches and arithmetic operations that are very energy demanding, and merely reducing the data size of inputs and parameters to meet power constraints might not be the optimal strategy due to significant impact on output error. Hence, this work proposes a framework for arithmetic circuit generation, enabling an architectural exploration that seeks to maximize as much as possible the energy efficiency. As a case study, this thesis also proposes a jointly optimized software-hardware approach to implement a neural network-based heart rate estimation application from photoplethysmogram signals. This approach combines binarization and quantization techniques to reduce computation requirements, making the model more suitable for hardware implementation. A custom hardware architecture is proposed for this application to achieve real-time operation with maximum energy efficiency. The stream-based architecture minimizes the system latency adopting a full pipeline implementation exploring the application requirements. This architecture was validated on both FPGA and ASIC platforms to ensure its feasibility on embedded devices.

Keywords: Neural networks. VLSI design. low power CMOS. hardware accelerator. heart rate estimation. PPG.

RESUMO

O aumento da densidade de potência e do uso pervasivo de aplicações com alto custo em esforço computacional e em dissipação de potência exigem eficiência energética no projeto CMOS. A busca por sistemas eficientes energeticamente é particularmente crítica em dispositivos vestíveis para monitoramento de sinais vitais uma vez que estes devem operar ininterruptamente mesmo com uma fonte de energia limitada disponível nas baterias miniaturizadas. Há um interesse crescente no emprego de aplicações baseadas em redes neurais para o processamento de dados em dispositivos embarcados. Redes neurais possuem estruturas inerentemente complexas para implementação, seja em *software*, *hardware* ou em uma combinação estreita de ambos. Tais redes requerem milhões de operações aritméticas e acessos à memória que demandam um gasto de energia elevado, e simplesmente reduzir a largura de representação dos parâmetros e dados de entrada para respeitar as restrições de dissipação de energia pode não ser a melhor estratégia devido ao impacto no erro percebido no resultado da aplicação. Assim, esse trabalho propõe um *framework* para geração de circuitos aritméticos, permitindo uma exploração arquitetura para buscar a máxima eficiência energética. Como estudo de caso, essa tese também propõe uma abordagem de otimização conjunta de *hardware* e *software* para implemetar um aplicação para estimação de frequência cardíaca a partir de sinais de fotopletismografia baseada em uma implementação de redes neurais. Essa abordagem combina técnicas de binarização e quantização para reduzir os requisitos de processamento, transformando o modelo em uma implementação mais adequada para a execução em *hardware*. Uma arquitetura de hardware customizada é proposta para esta aplicação para operação em tempo real com máxima eficiência energética. Esta arquitetura baseada em fluxo de dados minimiza a latência do sistema ao adotar uma implementação com *pipeline* em todos os estágios, explorando os requisitos da aplicação. Esta arquitetura foi validada em plataformas FPGA e ASIC para garantir sua viabilidade em sistemas embarcados.

Palavras-chave: Neural networks. VLSI design. low power CMOS. hardware accelerator. heart rate estimation. PPG.

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVIATIONS AND ACRONYMS

ASIC	Application-Specific Integrated Circuits
ANN	Artificial Neural Network
AV	Atrioventricular node
BPTT	Backpropagation Through Time
BN	Batch Normalization Layer
bCNN	Binary CNN
bCorNET	Binary CorNET
bLSTM	Binary LSTM
BLAS	Basic Linear Algebra Subprograms
BRAM	Block RAM
CGEN	Carry Generator Module
CLA	Carry Look-ahead Adder
CPA	Carry-propagating Adder
CPU	Central Processing Unit
CMOS	Complementary Metal-Oxide-Semiconductor
CEC	Constant Error Carousel
CNN	Convolutional Neural Networks
DNN	Deep Neural Networks
DL	Dense Layer
DUT	Design Under Test
DVFAS	Dynamic voltage-frequency-ccuracy Scaling
ECG	Electrocardiogram
FPGA	Field Programmable Gate Array
FA	Full Adder

GRU	Gated Recurrent Unit
GOPS	Giga-operations per second
GPU	Graphic Processing Unit
HR	Heart Rate
HLS	High-level Synthesis
HOG	Histogram of Oriented Gradients
IFM	Input Feature Map
LSB	Least Significant Bit
LOSO	Leave-one-subject-out strategy
LOWO	Leave-one-window-out strategy
LED	Light-emitting Diode
LSTM	Long short-term memory
LUT	Lookup Table
ML	Machine Learning
MAE	Mean Absolute Error
MSE	Mean Squared Error
MSB	Most Significant Bit
MA	Motion Artifacts
MLP	Multi-Layer Perceptrons
MAC	Multiply-and-accumulate
NFU	Neural Functional Unit
NN	Neural Networks
NL	Non-linear
OCR	Optical Character Recognition
OFM	Output Feature Map
PD	Photodetector

PPG	Photoplethysmography
PLE	Physically-Aware Layout Estimation
PE	Processing Element
PIM	Processing-in-Memory
QoR	Quality of Results
ReLU	Rectified Linear Unit
RNN	Recurrent Neural Network
RF	Register File
RCA	Ripple Carry Adder
RMSProp	Root Mean Square Propagation
SPC	Signal Processing Cup
SNR	Signal-to-noise ratio
SIMD	Single Instruction, Multiple Data
SIMT	Single Instruction, Multiple Thread
SA	Sinoatrial node
SRAM	Static Random Access Memory
SDF	Standard Delay Format
SDAE	Standard Deviation of the Absolute Error
SGD	Stochastic Gradient Descent
STE	Straight-through Estimator
SUT	Subject Under Test
SAIF	Switching Activity Interchange Format
SL	System Latency
TCF	Toggle Count Format
UVM	Universal Verification Methodology
VCD	Value Change Dump

VALU Vector Arithmetic and Logic Unit

CONTENTS

1 INTRODUCTION

In recent years, the push towards miniaturization in Complementary Metal-Oxide-Semiconductor (CMOS) devices led to the development of emerging applications like Internet of Things (IoT), large scale cloud computing, house, and automobile automation, and mobile computing. Each application has different complexity demands from the computing stack. Nevertheless, the underlying hardware has to cope with several challenges performance-wise, especially with respect to the ever-growing power density in these devices (DENNARD, 2015). Current general-purpose processors already run at frequencies that are managed or slowed to cope with very high power densities in silicon areas with high switching activity (SCHWIERZ; LIOU; WONG, 2010). This increased power dissipation leads to several problems to be addressed by chip, packaging, and cooling design solutions, as they impact both device reliability and system performance. These are known to degrade as the temperature rises (for voltage supplies well above the FET thresholds) due to the physical characteristics of the semiconductor devices.

Such semiconductor advancements improved the computation capability of edge devices, which led to the development of the *edge computing* paradigm where the tasks are executed locally on the device instead of exchanging the information with a powerful cloud system (LIANG et al., 2020). Processing data locally has several benefits, like reduced latency, improved security, and reduced data transmission energy, although the increased computing requirements compromise part of these energy savings demanded from the edge devices.

This case is perfectly illustrated by deep learning applications, which usually require an immense compute power even in such edge devices. Image and speech recognition, language processing, signal processing, and industrial plant monitoring are typical applications present in a plethora of edge devices (DE SILVA et al., 2020). In these applications, edge devices are only concerned about the inference part of the algorithm as the training process is executed in large data centers. Nonetheless, the inference task computational cost is hardly met by general-purpose processors, limiting their applications.

A solution to this problem is to adopt a hardware-software co-design optimization approach to reduce network complexity and to offer hardware support for optimized data flow (VERHELST; MOONS, 2017). On the one hand, reducing the complexity is quintessential as some neural networks may achieve 100 giga-operations (GOPS) per each evaluation. On the other hand, these operations make extensive use of arithmetic circuits

and data movement procedures, two aspects that are directly linked with the network architecture (ZHOU et al., 2019). This approach has fostered research groups to investigate these optimizations for general-purpose circuits able to accelerate a multitude of neural networks.

1.1 Problem formulation

Designing dedicated hardware architectures for neural networks can bring together optimizations from both the software and hardware level. The use of efficient hardware accelerators enables complex applications such as object recognition and signal processing to operate reliably in edge devices.

The medicine evolution changed the actuation paradigm to be more focused on disease prevention and early detection as the scarce resources are better distributed this way. This approach was mainly possible with wearable health-monitoring devices that can obtain a humongous pile of data that can be processed and presented as useful information for both patients and health service workers.

The most critical information gathered by these devices is the heart rate and its variability. It not only helps athletes improve their performance, but it also helps to monitor and detect heart-related diseases that are usually silent and likely to be fatal. Photoplethysmography (PPG) has arisen as a low-cost, non-intrusive heart rate monitoring technology that has been embedded in smartwatches and fitness trackers.

Nonetheless, wrist-worn PPG devices are heavily affected by motion artifacts, which can distort the signal quality, reducing the measuring effectiveness of the system. In that sense, neural networks are a promising alternative to traditional signal processing algorithms as they usually can extract more information from a single signal, reducing the system complexity. Nonetheless, most devices available on the market could only process these networks using general-purpose processors, which is not only energy-inefficient for this task, but additionally might not be able to process it in real-time.

1.2 Thesis claim and objectives

This thesis claims that efficient neural network inference can be achieved by adopting a hardware-software joint optimization approach, enabling real-time operation in em-

bedded devices with limited compute capability and power availability. The following set of objectives was set to accomplish these claims:

- Provide a literature review on neural networks, considering the theoretical foundations along with techniques to cope with the complexity of such algorithms on both algorithmic and circuit levels.
- Propose a framework for arithmetic core generation to explore the implementation of efficient arithmetic operators.
- Explore the inherent time dependency characteristic of recurrent neural networks for heart rate estimation from a unidimensional signal.
- Provide a co-design strategy to obtain an optimized recurrent neural network model that is able to run on a dedicated hardware platform for heart rate inference from photoplethysmography signals.
- Propose a hardware architecture that employs energy-efficient arithmetic circuits and low-power techniques, targeting both FPGA and ASIC platforms to achieve real-time heart rate inference based on the proposed recurrent neural network model.

The main motivation relies on the fact that neural networks often have hundreds of thousands of parameters, and a single inference may require millions of operations. Squeezing every bit of performance with the smallest power dissipation is fundamental in energy-constrained devices. Therefore, this work proposes a cross-layer optimization from the network down to the arithmetic circuits that constitute the heart monitoring system.

1.3 Thesis Organization

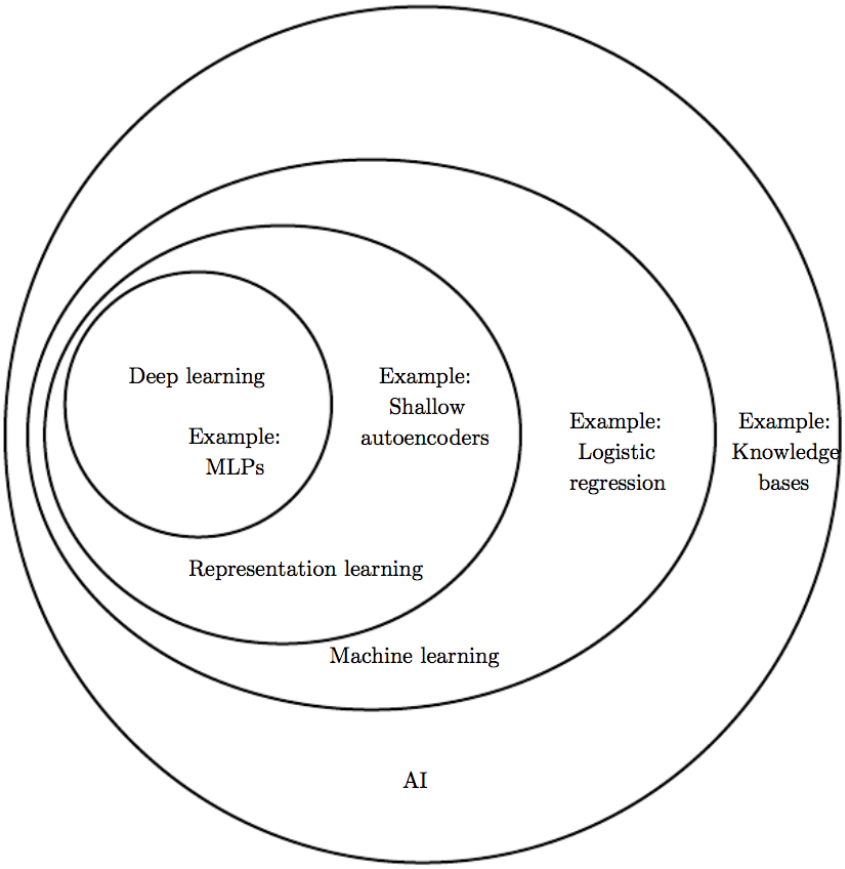
The remaining of this thesis proposal is organized as follows: a comprehensive background of neural networks is presented in Chapter 2. Chapter 3 explores optimization techniques to cope with the high complexity of neural networks and general-purpose and custom computational platforms. A review of arithmetic circuits is presented in Chapter 4. Chapter 5 describes RTLGen, a Python-based framework for arithmetic circuit generation. A brief overview of heart rate estimation is presented in Chapter 6, along with the structure of CorNET, a deep neural network for heart rate estimation from wrist-worn PPG signals. The binarization and custom hardware implementation of the CorNET framework are presented in Chapter 7. Finally, Chapter 8 summarizes the conclusion of this thesis, along

with the directions of future work. A list of all publications accomplished by the author during the years of this Ph.D. research is presented in Appendix A.

2 OVERVIEW OF DEEP LEARNING

Machine Learning (ML) comprises a wide variety of subareas that have one common factor: they all use mathematical models and a huge amount of data to classify, predict or generate new information that can be useful to the application they were conceived for (GOODFELLOW; BENGIO; COURVILLE, 2016). ML algorithms primary role is to obtain knowledge about a given set of inputs based on a model previously built from observations. All ML systems aim to combine a set of inputs in a specific manner to produce a useful prediction on data that has never been applied to the model. From a mathematical point of view, these inputs are variables, also known as features. They are linearly combined with weights and biases to produce a prediction about a particular characteristic of the input. The whole Machine Learning concept can be divided into several classes, as shown in Figure 2.1.

Figure 2.1: Artificial intelligence subareas division



Source: (GOODFELLOW; BENGIO; COURVILLE, 2016)

A machine learning algorithm can be seen as a program that learns "from an experience E with respect to some class of tasks T and a performance measure P, if its per-

formance at tasks in T, as measured by P, improves with the experience E" (MITCHELL, 1997, p. 2).

The **Task T** determines what the application goal is. According to (GOODFELLOW; BENGIO; COURVILLE, 2016, p. 97),

> Machine learning tasks are usually described in terms of how the machine learning system should process an example. An example is a collection of features that have been quantitatively measured from some object or event that we want the machine learning system to process.

There is a multitude of tasks that can be accomplished with ML. For instance, *classification* tasks aim to detect and specify to which category or class the current input belongs. They fostered several competitions like image classification and face recognition, among others. Particularly, the ImageNet Large Scale Visual Recognition Challenge (RUSSAKOVSKY et al., 2015) played a fundamental role in this field as it led to the development of new deep learning techniques and extremely accurate neural networks. Another promising task type is *transcription*, where the ML algorithm converts the input into another representation, like speech recognition, license plate reading, and optical character recognition. A non-exhaustive task list includes anomaly detection (e.g., fraud detection), structured output (e.g., image segmentation), machine translation (e.g., automatic language translation), regression (e.g., heart rate estimation), among others.

Regardless of the task that must be accomplished by the ML algorithm, its **Performance P** must be measured to indicate the algorithm effectiveness. The performance metric is directly dependent on the task being carried by the system. Generally, in classification and transcription tasks, P is dictated by the accuracy (or error rate), a metric that indicates the proportion of correct outputs. However, the most important P measurement is obtained when the algorithm is evaluated on a *test dataset*. It contains data that the model has never seen before as it determines the generalization capability of the model and how well it performs in a real-world scenario (GOODFELLOW; BENGIO; COURVILLE, 2016).

The algorithm performance P, for a given task T, improves according to its **Experience E** during the learning process. The algorithm can adopt either an *unsupervised* or *supervised* learning process, which depends on the type of experience available. There are other learning approaches like *reinforcement* learning, which are not the focus of this work.

- **Unsupervised learning:** the algorithm learns useful properties and characteristics of the structure of the dataset. In this case, no labels, targets or outcomes are spec-

ified *a priori* for each member of the dataset, so the model must make sense of the input data without supervision. Examples of this approach are k-means clustering, principal component analysis, and outlier detection.

- **Supervised learning:** the experience available to the algorithm includes input data with associated labels or targets. In this case, the algorithm learns to predict an output $\mathbf{y}$ from input $\mathbf{x}$. Examples of this approach include image classification (like ImageNet and the Iris dataset), handwritten recognition, and speech recognition.

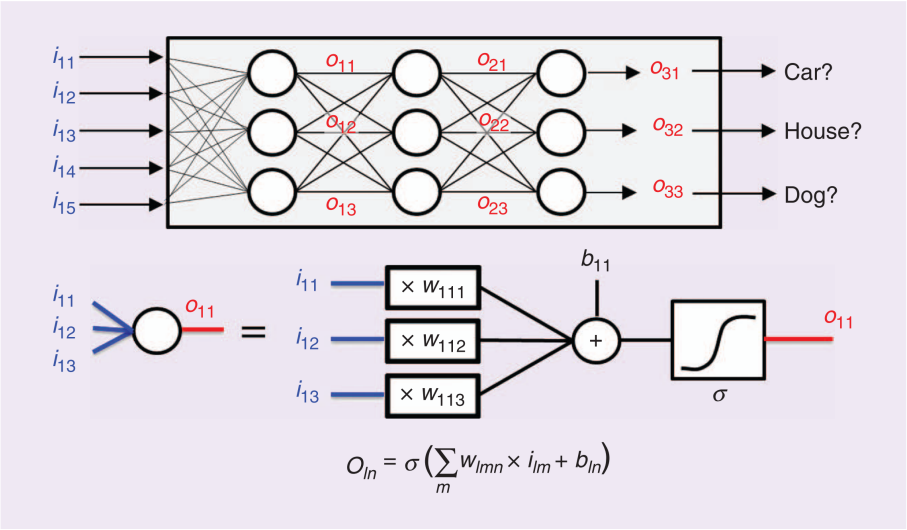
2.1 Deep Learning Algorithms

Simple machine learning algorithms are not suited to solve complex tasks like object and speech recognition, requiring more advanced techniques that improve the model ability to generalize in these tasks (GOODFELLOW; BENGIO; COURVILLE, 2016). This challenge fostered the development of deep learning techniques based on a deep graph with many layers representing a model, describing reality through a hierarchy of concepts (MOONS; BANKMAN; VERHELST, 2019). Notably, these techniques are implemented with neural networks (NN), which are able to solve subjective problems, like speech recognition and image classification, through a learning methodology.

Neural networks are inspired on the brain biology, as the latter can acquire several sensorial data – vision, sound, and others – and process them in an array of interconnected neurons that generates a response for these stimuli. The intricate connections among neurons give the brain the ability to perform various tasks like recognizing some face in a photo, understanding spoken words, and so on. Although these tasks are trivial for humans, they are not easily accomplished by computers when traditional software is used.

This class of algorithms has been studied for a long time aiming to mimic the brain's ability to process input data through neurons to generate a stimuli response (ROSENBLATT, 1958; MARBLESTONE; WAYNE; KORDING, 2016; WIDROW; LEHR, 1990). These neural networks were initially based on the McCulloch–Pitts neuron (MCCULLOCH; PITTS, 1943), which featured multiple inputs, and each one was associated with a specific weight to control the influence of such inputs, as illustrated in Figure 2.2. These inputs are linearly combined before passing through a non-linear function – known as the activation function – to simulate the synapse in a brain neuron. Often, non-linear functions – like a sigmoid, hyperbolic tangent, rectified linear unit (ReLU) – are used as the activation function.

Figure 2.2: Example of an artificial neural network and artificial neuron.



Source: (VERHELST; MOONS, 2017)

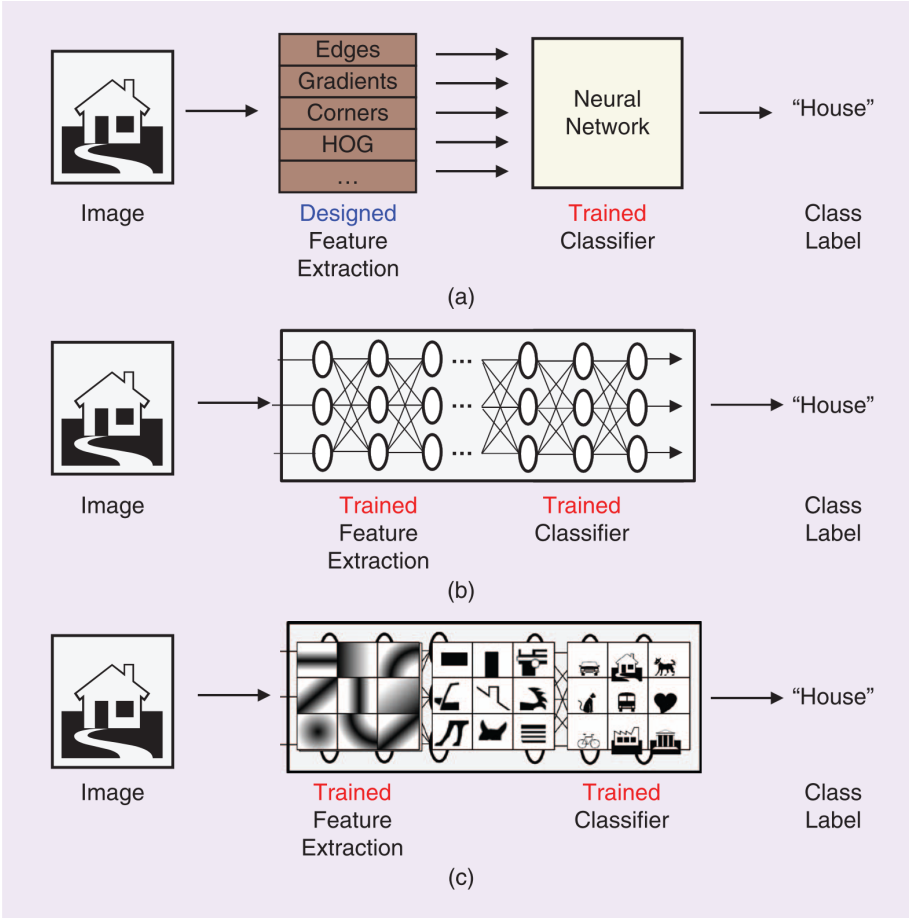
Neural network models consist of several layers of neurons, and each neuron of a given layer has a weighted connection to each neuron of the next layer. Also, they often have a bias value, which enables the translation of the activation function. The activation function must be implemented as a non-linear function to make possible learning more complex featuresand allow the training algorithms to work properly. In the end, the combination of these connections leads to an expected output according to algorithm objectives. In class-based algorithms like image recognition, the output indicates the probability of the current input to belong to a given class (see Figure 2.2). Both weights and biases of neurons in a network describe the network parameters, and the definition of the network structure is tailored for each target application.

First neural networks were conceived as classifiers whose inputs were obtained through hand-crafted feature extractors that could translate the raw data (like a two-dimensional array of pixel values) into an appropriate representation. They required that application domain experts expend considerable effort in properly selecting and implementing the most suitable feature extractors, including edge detection filters, histogram of oriented gradients (HOG), among others (LECUN; BENGIO; HINTON, 2015). This process was known as *conventional machine learning*, and its structure is presented in Figure 2.3a.

Nevertheless, these models were limited to small neural networks with a limited number of layers and neurons, which severely capped the ability of the model to learn more complex features. According to (SZE et al., 2017b), such layer limitation was due

to three main reasons: (a) larger networks had convergence issues due to the training algorithm available at that time, (b) datasets were not large enough, and (c) computational power of the machines in the past could not cope with the complexity of such models. Despite these drawbacks, these models were state-of-the-art in several applications with high performance in object recognition tasks like optical character recognition (OCR).

Figure 2.3: Evolution of machine learning implementation flow



Source: (VERHELST; MOONS, 2017)

As the technology evolved, providing more computing power and more massive datasets became available, these networks were able to stack more layers and neurons. Hence, a key paradigm shift happened with the adoption of *representation learning*, where the neural network model was responsible for both the feature extractor and the classifier (Figure 2.3b). According to LeCun, Bengio and Hinton (2015, p. 436), "representation learning is a set of methods that allows a machine to be fed with raw data and to automatically discover the representations needed for detection or classification."

Deep learning algorithms employ multiple representation levels by stacking non-linear modules that progressively transform the raw representation at one level into a more

abstract representation on the next level. This approach allows the network to learn very complex functions. The key benefit of deep learning is that all features are learned directly from data based on a learning procedure, removing the need for hand-crafted or previous, specific, and separate software pre-processing for feature extraction entirely. For instance, Figure 2.3c shows the inspection of a given trained deep learning network where the initial layers detect simple features like lines and color gradients. As the network deepens, more complex features are detected like a house or a car. In Olah, Mordvintsev and Schubert (2017), the authors developed a feature visualization tool that allows neural network developers to investigate the patterns learned by each module in each layer for a given model.

This new approach for constructing neural networks excels at discovering complex structures in high-dimensional data, and it led to models that beat previous records in object and image recognition, molecular activity prediction, and other applications in vast domains of science, technology, and business (LECUN; BENGIO; HINTON, 2015). Recent advances showed that neural networks have an outstanding performance that even surpassed human abilities. For instance, He et al. (2015) shows the first NN architecture with better accuracy than the human level for a 1000-class image classification dataset. The authors in Xiong et al. (2016) proposed a NN for conversational speech recognition that achieved the same error rate of trained professionals transcribers. However, these achievements come at a high cost in algorithm complexity and computational requirements. Powerful neural networks have hundreds of thousands or even millions of parameters used in millions of operations, posing considerable challenges in embedded platforms that are energy-constrained with limited computing capabilities like smartwatches, smartphones, and drones. For further information, a comprehensive review of the deep learning evolution is presented in Wason (2018), indicating each algorithm's goal and application domain.

Regardless of the neural network application, they all undergo two procedures: training and inference phases. During the training phase, a set of inputs is fed to the model, and, according to the learning approach, the training algorithm updates the weights accordingly so the outcome will be as expected. This automated process is the backbone of all ML algorithms. Since it demands an enormous amount of data, this procedure usually occurs offline, i.e., the network training is executed in powerful machines outside the real scenario the network will be deployed. Depending on the number of parameters and input datasets, this training phase goes from minutes, in small models, up to weeks

in deeper networks. Usually, once the model achieves satisfactory accuracy, the network refinement stops, and the network is deployed to perform its target application. Hence the time, bandwidth, and consumed energy on the training process are not particularly concerning due to its single-event nature, even though the desired model must feature such characteristics. Since this work focuses on supervised learning application, Section 2.2 further discusses the processes involved in this learning approach.

Once the training process is done, the model can perform its task by computing the network output based on the inputs and weights. This process is known as *network inference*. Given the current constraints in processing time, transmission bandwidth, information security, and power consumption in mobile devices like sensors, smartphones, smartwatches, etc., the current trend is to embed the inferece process near the sensor. Several studies address the advantages of using this approach, like in (DU et al., 2015; CAVIGELLI et al., 2015).

2.2 Supervised Learning

The main point of deep learning algorithms resides in the self-learning methodology, where the network adapts itself without the need for tailored filters and expert-based algorithms to solve a problem. If a large amount of data is given to the network, it will learn how to adapt its mathematical model to represent the target application (GÉRON, 2017). This task can be achieved in many ways, yet this work is limited to the supervised learning approach.

Training a neural network is an optimization process that involves maximizing or minimizing some function that indicates the model performance. During training, the data is usually split into training and test datasets. The former usually contains an enormous amount of data, and it is used to train the network parameters (weights and biases). On supervised learning, this dataset may include a label associated with each dataset element depending on the training algorithm. This label corresponds to the expected network output when the information is fed to the algorithm. Conversely, the test dataset is used to evaluate the algorithm performance on data that it has never seen before.

Although the primary mechanism of supervised training is the same for all ML-based algorithms, there is a fundamental difference between linear models and neural networks. The latter introduces non-linearities, transforming the learning process into a non-convex optimization problem (GOODFELLOW; BENGIO; COURVILLE, 2016).

This issue is usually addressed using a gradient-based optimizer since other methods do not scale for neural networks.

2.2.1 Loss Function

In all gradient-based optimizers, the main goal is to minimize a loss function (also known as criterion or cost function). Hence, choosing an adequate loss function is quintessential to train a neural network successfully. These functions compute the metric that indicates how well the model fits within the training datasets. Choosing the most suitable loss function depends on the type of problem that the neural network is supposed to solve, although it must be differentiable to allow the gradient computation (GOODFELLOW; BENGIO; COURVILLE, 2016). In summary, loss functions have two roles: (a) they penalize the model complexity to some extent, and (b) they measure the compatibility between the input and the ground truth. The data loss is taken as the average loss over all the N individual examples, as shown below:

$$L = \frac{1}{N} \sum_{i=0}^{N-1} L_i \tag{2.1}$$

These functions are often divided into two categories. The regression loss functions predict quantities, and they suit applications like the estimation of the product price, prediction over yearly sales in a company, etc. Classification loss functions predict labels, and they are more appropriate for classification problems where the output value should correspond to a limited set of classes. This type of loss function is commonly seen in image classification networks as the possible network outcomes are discrete and well-defined (MOONS; BANKMAN; VERHELST, 2019).

In classification networks, the network is trained to maximize the likelihood of the predicted value with respect to the label associated with that input. Therefore, the cost function is given by the negative log-likelihood, equivalent to the cross-entropy between the training data and the model distribution (GOODFELLOW; BENGIO; COURVILLE, 2016). The cross-entropy (H) for N neurons on the output layer is computed according to (2.2) where $p(x)$ indicates the distribution for each true label, and $q(x)$ represents the distribution for each predicted label. This loss function is usually combined with a softmax function, so the output is a 1-hot encoded vector distribution p, where 1 is at the index of the true label, and 0 otherwise.

$$H(p,q) = -\sum_{x}^{N} p(x)\log\left(q(x)\right) \tag{2.2}$$

Conversely, regression-based networks operate on real-valued quantities, so the loss function must compute the error between the predicted output and the expected value. The most common approaches are the mean squared error (MSE), also known as L_2-norm, and the mean absolute error (MAE), also known as the L_1-norm. These functions were once used for classification tasks, but they are no longer adopted for that task type. Both functions compute average error difference between the predicted output and the golden label for all N neurons on the output layer, as stated below:

$$MSE = \frac{1}{N}\sum_{i}^{N} |p_i - y_i|^2 \tag{2.3}$$

$$MAE = \frac{1}{N}\sum_{i}^{N} |p_i - y_i| \tag{2.4}$$

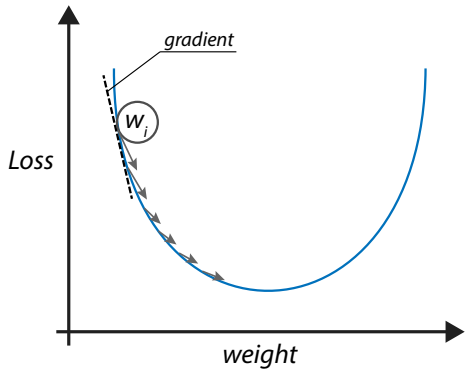
In both functions, p_i and y_i are the predicted and expected outputs. The MSE approach is more stable and computationally efficient than the MAE. However, MSE is less robust than the MAE as it severely penalizes the eventual outliers, which may lead to a model more optimized for this case instead of the general case.

2.2.2 Backpropagation Algorithm

Once the loss function is determined, the partial derivatives of this function must be computed with respect to all weights and biases through the chain rule of differentiation. The *backpropagation algorithm* is an efficient approach to accomplish this task, and it sequentially computes the derivatives from the output of the last network layer up to its inputs (SZE et al., 2017b). The backpropagation algorithm computes the negative gradient of the loss function to determine the direction in which the error decrease according to Figure 2.4.

Note that the backpropagation is only part of the learning process as its role is computing the error gradients without any weight modification (GOODFELLOW; BENGIO; COURVILLE, 2016). In that sense, the algorithm looks for all partial derivatives of

Figure 2.4: Weight update using the gradient descent approach



the loss function L to any weight w_{ij} which connects the neuron j to a previous neuron i throughout the entire network. Assuming that f_k is an output activation function that transforms the output o_k into the expected form, the general computation can be defined as:

$$\frac{\partial L}{\partial w_{ij}} = \sum_p \left[\frac{\partial L}{\partial f_p} \left(\sum_k \frac{\partial f_p}{\partial o_k} \frac{\partial o_k}{\partial w_{ij}} \right) \right] \tag{2.5}$$

In (2.5), the term $\sum_p$ is the summation of all outputs and $\sum_k$ is summation of all inputs that affect each output o_p. Since the partial derivatives must be computed for all network nodes, this process has considerable memory usage and compute requirements, limiting its integration in embedded devices for complete on-edge training. An extensive mathematical explanation of the backpropagation algorithm is presented in (GOODFELLOW; BENGIO; COURVILLE, 2016).

2.2.3 Optimization algorithms

As stated earlier, the learning process in a neural network is a non-convex optimization problem since the partial derivatives computed during the backpropagation may lead to multiple critical points – minima, maxima and saddle points – which may severely limit the learning performance of the network. In most cases, the network optimizers are based on a hill-climbing algorithm known as *gradient descent*, which aims to minimize the error by updating the parameters according to the negative value of the error gradient (LECUN; BENGIO; HINTON, 2015).

The gradient descent algorithm works as follows. Assuming **w** is a vector containing the weights that describe the connections in a neural network, the update procedure follows the Equation 2.6. The current weights change accordingly to the update factor $\Delta\mathbf{w}$.

$$\mathbf{w} = \mathbf{w} + \Delta\mathbf{w} \tag{2.6}$$

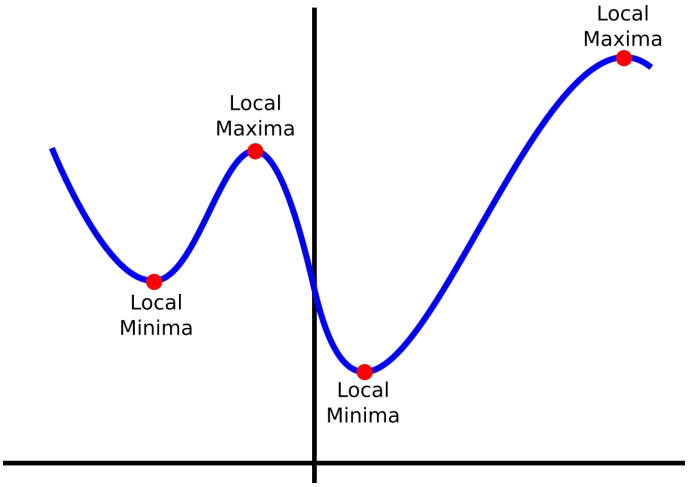
where the update factor is calculated individually for each weight based on the gradient estimate (g) for that weight:

$$\Delta w_j = -\eta \frac{\partial J}{\partial w_j} = -\eta g \tag{2.7}$$

The update factor depends on the gradient of the loss function with respect to the current weight. Also, the η factor is the learning rate, and it controls the update step size for the new weight. Choosing the right value for the learning rate is not straightforward for two reasons:

- If η is too small, the learning process slows down as it will require more steps. Further, since it is possible to have multiple local minima (see Figure 2.5, a small learning rate may leave the update process in a non-optimal minima point.
- If η is too big, the optimizer may never reach a minima, and the model can even diverge.

Figure 2.5: Multiple local minima and maxima points in a loss function



Despite the effectiveness of the gradient descent algorithm, it has a poor performance on deep neural networks due to the enormous computational effort to advance a

training epoch with respect to the network learning process. Here, the term *epoch* indicates the number of times the optimizer has passed through all values on the training dataset. It takes a considerable amount of time to complete the network optimization as it requires the application of the entire dataset accumulating the loss, so it would be able to compute the gradients to update the weights.

The first alternative to this approach is the stochastic gradient descent (SGD) algorithm, proposed by Lecun et al. (1998), which is an approximation of the basic gradient descent algorithm. In this strategy, a minibatch of m samples is stochastically taken from the training dataset containing $N >> m$ samples and fed to the network. Then, the average loss for this minibatch is computed, and the weights are updated according to (2.6). This process is repeated until all N samples on the input dataset have been fed to the network. At this moment, the algorithm moves to the next training epoch. There are several benefits in using SGD, among them:

- **Scalable computation time:** The processing time between network parameter updates depends solely on the size of the minibatch regardless of the total dataset size. This approach ensures the convergence even when the training dataset is huge.
- **Noise introduction:** for each batch, the SGD estimator introduces a source of noise due to the stochastic characteristic of minibatch sampling. It prevents the convergence to local minima, although this could also lead to a non-convergent network.

There are several other variants of the SGD algorithm to improve the convergence rate, training time, and so on. Some of the most popular versions are listed below:

- **Momentum:** is an SGD extension designed to accelerate the learning process. It accumulates an exponentially decaying moving average of past gradients and keeps moving in that direction. In this case, the update process determined in (2.6) changes to $w = w + v$ where v is the *momentum* or velocity and it is given by $v = \alpha v - \eta g$. Here, α is hyperparameter which determines "how quickly the contributions of previous gradients exponentially decay" (GOODFELLOW; BENGIO; COURVILLE, 2016, p. 293).
- **AdaGrad:** is one of the first learning algorithms with an adaptive learning rate. Here, each model parameter has a specific η value that is updated in each minibatch. The learning rate update is inversely proportional to the square root of the sum of all the historical squared values of the gradient (DUCHI; HAZAN; SINGER, 2011). In this algorithm, the learning rate decreasing speed is directly proportional to the

gradient estimate's magnitude for that given parameter.

- **Adam:** is a combination of the RMSProp (a variant of momentum) and AdaGrad algorithms (KINGMA; BA, 2014). This method computes the adaptive learning rates using two momentum estimates of the gradients, hence the name Adam (adaptive moment estimation). The intrinsic gradient rescaling and adaptive momentum leads to an algorithm fairly robust to initial hyperparameter definitions (GOODFELLOW; BENGIO; COURVILLE, 2016).

2.3 Feed-forward Networks

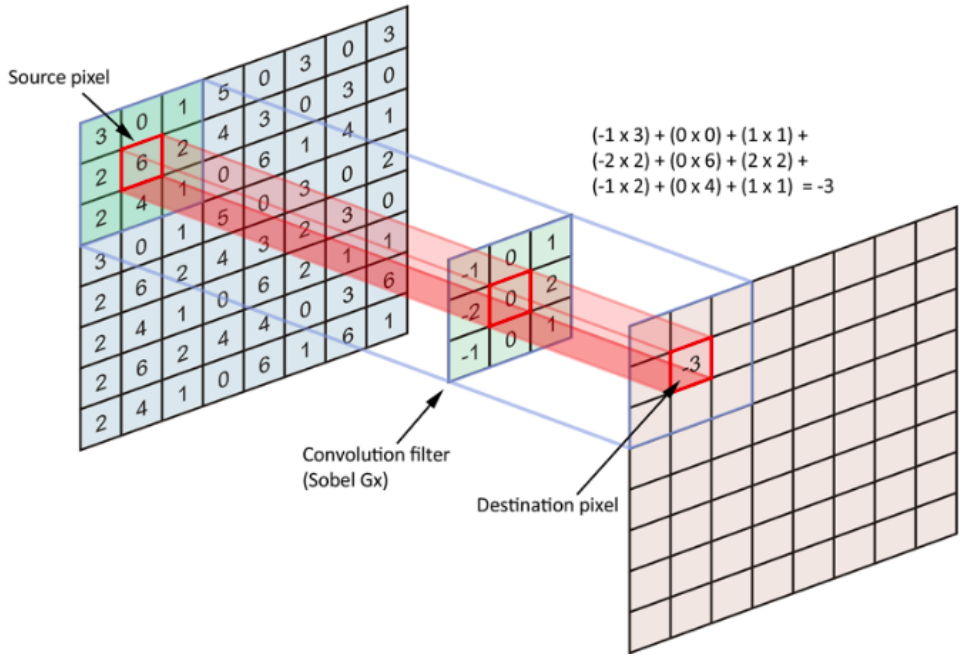
Feed-forward networks are also known as multi-layer perceptrons (MLPs), representing the first type of artificial neural network (ANN) ever proposed. These models do not have any type of internal feedback connections, so previous computations do not influence the current output result. Traditional MLPs were composed of fully-connected layers where each neuron on a given layer is connected to all neurons on the next layer, as exemplified in Figure 2.3b. The goal of a feed-forward network is to approximate a function f that maps the input **x** to a desired output **y** (GOODFELLOW; BENGIO; COURVILLE, 2016). For a n layer network, this function is defined as $f(\mathbf{x}) = f^n(f^{n-1}(...f^2(f^1(x))))$, where f^1 is the *network input*, $f^2()$ to $f^{n-2}()$ are the *hidden layers*, and $f^{n-1}()$ is the *output layer*. The number of neurons on a given layer determines the layer *width*, and it could vary between layers within the same model.

According to Sze et al. (2017b), deep neural networks (DNN) are so-called whenever they "have more than three layers, i.e., more than one hidden layer." However, models based only on fully-connected layers do not scale very well due to the number of parameters that need to be trained. For instance, assuming a DNN with L layers, each one of them with N neurons, there would be $L.(N^2 + N)$ that must be trained. Considering an image recognition task where each input can have thousand of pixels, the number of parameters rapidly explodes. The breakthroughs in deep learning also led to new network topologies that addressed this issue.

2.3.1 Convolutional Neural Networks

Convolutional neural networks (CNNs) are a particular type of feed-forward neural network where the neural connections are limited, addressing some of the problems of DNNs. The virtual cortex on the human brain inspires the foundation of these CNNs as the former has several localized receptive fields whose neurons only produce spikes when there are stimuli on specific locations (LECUN; BENGIO et al., 1995). According to Goodfellow, Bengio and Courville (2016, p. 326), CNNs are "simply neural networks that use convolution in place of general matrix multiplication in at least one of their layers." These networks have several CNN layers cascaded, which are trained to represent hierarchical features.

Figure 2.6: Example a 3×3 2-D convolution over a single input channel

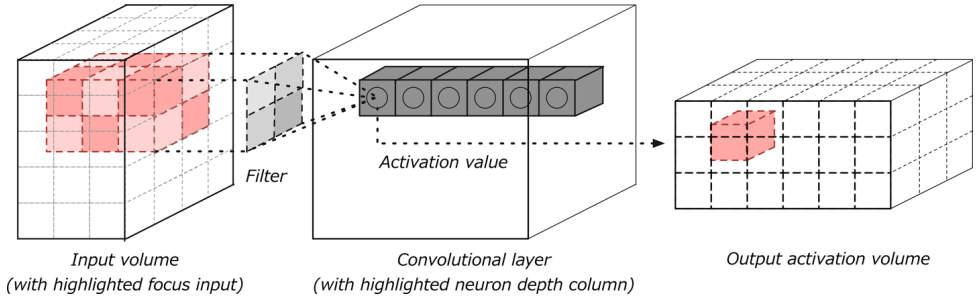


Source: (CORNELISSE, 2018)

Convolutional layers implement sets of filters with limited neural connections that convolve with the input feature maps (IFMs) to produce the output feature maps (OFMs). Each filter corresponds to a learned patch that identifies a specific feature. The filter size determines the neuron's local receptive field, indicating the amount of information a neuron will receive to combine and produce an output. Figure 2.6 illustrates how a 2-D convolutional layer work considering a filter size of 3×3. The grid in blue is the input image while the small grid in green represents the filter coefficients – also known as kernel – found during training. Note that this filter is projected over the input feature

map, and only those input values are considered for computing the output value on the output feature map.

Figure 2.7: View of a 2D convolutional layer with multiple input channels



Source: (GIBSON; PATTERSON, 2017)

Each CNN layer may have multiple filters to learn specific features that will generate multiple channels on the output feature map. Whenever the input feature maps have more than one channel, each filter is composed of a filter stack, one for each input channel, whose final value is the sum of the convolution across all the channels, as shown in Figure 2.7. In this example, the input is a 3-channel 2-dimensional IFM, and each filter can be seen as a single 3-D computation unit. The formal mathematical description to compute the outputs of a filter f on the layer l is given by:

$$O[f][x][y] = \sum_{c=0}^{C-1} \sum_{i=0}^{K-1} \sum_{j=0}^{K-1} I[c][Sx+i][Sy+j] \times W[f][c][i][j] + B[f] \tag{2.8}$$

In Equation 2.8, f is bounded by the number of filters on the layer (F), and $K \times K$ gives the filter size. The variables x and y indicate the position where the computation value will be stored, and they are limited by the size of the output feature map. O, I, W and B are the containers of the OFM, IFM, weight matrix and filter bias, respectively. The filter stride S sets how the kernel will slide over the feature map. If the stride is smaller than K, two consecutive filter windows will share, at least, K values. Increasing the value of S reduces the spatial dimension of the output feature maps as the inputs will undergo a subsampling. In most cases, the stride value is set to 1, although some implementations stride equal to 2. Table 2.1 summarizes the shape parameters of a given CNN layer.

Convolutional layers profit from three fundamental principles that allow such improved performance with lower complexity. The first one is *sparse connectivity* since the convolution operation employs kernels smaller than the inputs instead of the dense matrix multiplications in MLPs. These smaller patches are more suitable to detect sim-

Table 2.1: Shape parameters on a CNN layer

Parameter	Description
K	Width/height of the filter kernel
C	Number of input channels
F	Number of filters in the layer
M	Width/height of the output feature map
S	Convolution stride

pler yet meaningful features such as edges. Further, it requires fewer parameters and computing operations, and it improves the layer statistical efficiency (GOODFELLOW; BENGIO; COURVILLE, 2016). The second fundamental principle is *parameter sharing* as the same pattern learned by a given filter kernel can appear anywhere on the input feature map, so only one set of weights is required, which effectively reduces the model size (SZE et al., 2017b). Finally, the last principle is *equivariance*, which states that whenever an input suffers any type of modification, the output will change accordingly. For instance, in a time series application, any time event on the input will also appear on the output, although it would be time-shifted.

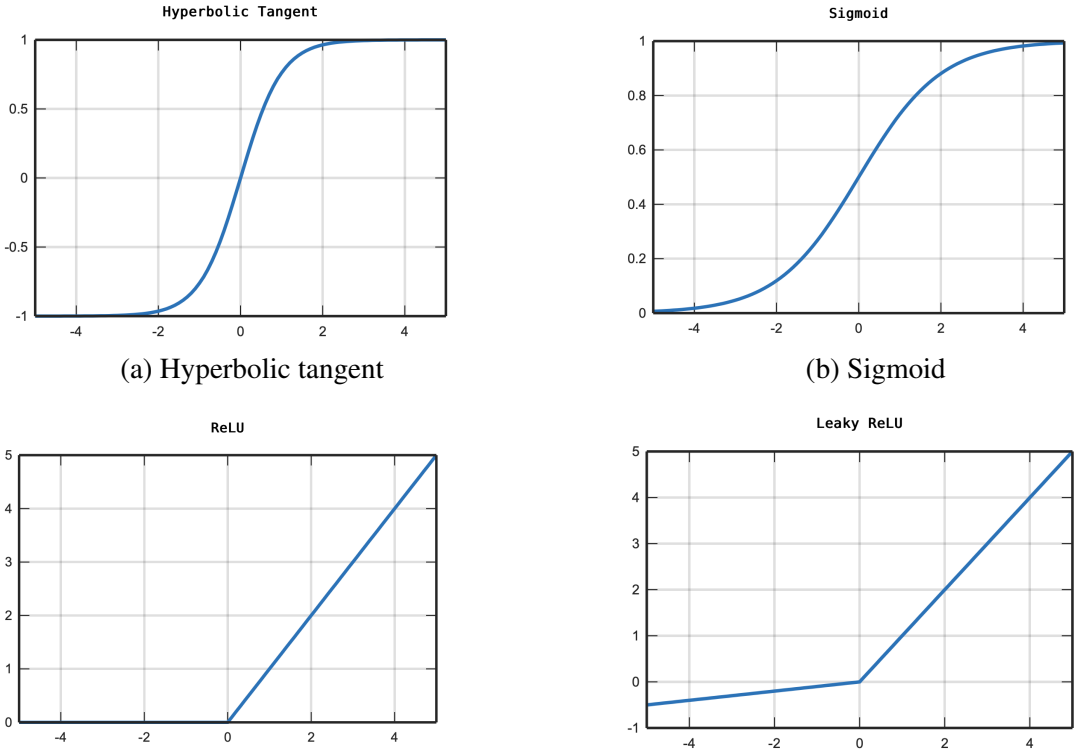
2.3.2 Activation Layer

In the biology world, a neuron only fires a synapse if the combination of its inputs is higher than a predefined threshold. Then, this synapse would be propagated to the neighbor neurons, and so on. This threshold-based decision to fire or not a neural synapse can be modeled as a non-linear function. Further, if only linear combinations are used throughout a neural network, the model will not be able to represent several geometric shapes and real-world problems. For a practical illustration of this problem, no linear combination can describe the XOR function. Thus, for a neural network to represent more complex features, it must include at least one non-linear layer (GOODFELLOW; BENGIO; COURVILLE, 2016).

Several activation functions have been proposed to add the required non-linearity to a neural network. Both sigmoid (Figure 2.8b) and hyperbolic tangent (Figure 2.8a) functions have been widely used since their outputs fall between the range $[0, 1]$ and $[-1, 1]$, respectively. This characteristic is desired for softmax classifiers used on multiclass classification problems.

Nair and Hinton (2010) showed that using approximate versions of these functions

Figure 2.8: Non-Linear Activation Functions



(a) Hyperbolic tangent

(b) Sigmoid

(c) Rectified Linear Unit

(d) Leaky Rectified Linear Unit

Source: The Author

improves system performance. The authors proposed the Rectified Linear Unit (ReLU), shown in Figure 2.8c, an almost-linear activation function where all negative values are set to zero. Due to the unbound output interval, this function avoids the early saturation of the outputs, and it keeps most of the advantages of linear models, resulting in lower training times and faster convergence (KRIZHEVSKY; SUTSKEVER; HINTON, 2012; GOODFELLOW; BENGIO; COURVILLE, 2016). Leaky ReLU (Figure 2.8d) solves the vanishing negative values problem of the traditional ReLU approach, which may affect the model ability to learn.

2.3.3 Pooling Layer

As the network deepens, the features found by each convolutional layer become more complex and more sensitive to input variations. Further, the number of filters within a convolutional layer usually increases towards the network output, leading to a ludicrous number of parameters to describe each layer.

The pooling layer reduces the spacial size of feature maps before their connections

subsequent layer within the network. The reduction is similar to a subsampling approach in signal processing and offers two key advantages. First, it effectively reduces both the number of computations and the number of parameters on the network. Second, it also controls the network overfitting, i.e., it prevents the model to become overly adjusted for the training dataset which leads to a poor performance on data never seen before. The subsampling introduces noise, leaving the model more permissive to variations on the input data (YU et al., 2014).

Figure 2.9: Multiple pooling algorithms for a 2×2 analysis window

Source: (SUN et al., 2020)

Figure 2.9 shows multiple pooling algorithms for an analysis window of 2×2. In pooling layers, the stride is equal to the size of the analysis window. According to the pooling approach, the algorithm will compare the values within the window and select the minimum, the average, or the maximum pixel values. Then, the output will be forwarded to the next layer to proceed with the computation.

2.3.4 Batch Normalization Layer

Each minibatch of the SGD-based learning algorithm changes each layer's input distribution, which may negatively impact the network training. This issue, also known as *covariate shift*, can be mitigated by introducing a batch normalization (BN) layer, which normalizes the inputs according to a running mean and variance (IOFFE; SZEGEDY,

2015). This data normalization is computed as:

$$y = \frac{x - \mu}{\sigma^2 + \epsilon}\gamma + \beta \tag{2.9}$$

The BN layer has four parameters. The pair (μ, σ) represents the batch mean and variance, respectively, and they are needed to obtain zero mean and unit variance on the inputs. Conversely, γ and β are scaling and shift parameters that modify the normalized value. If the BN is placed after a convolutional or fully-connected layer, its computation can be embedded into the weights of those layers, removing the need for additional computations (SZE et al., 2017b).

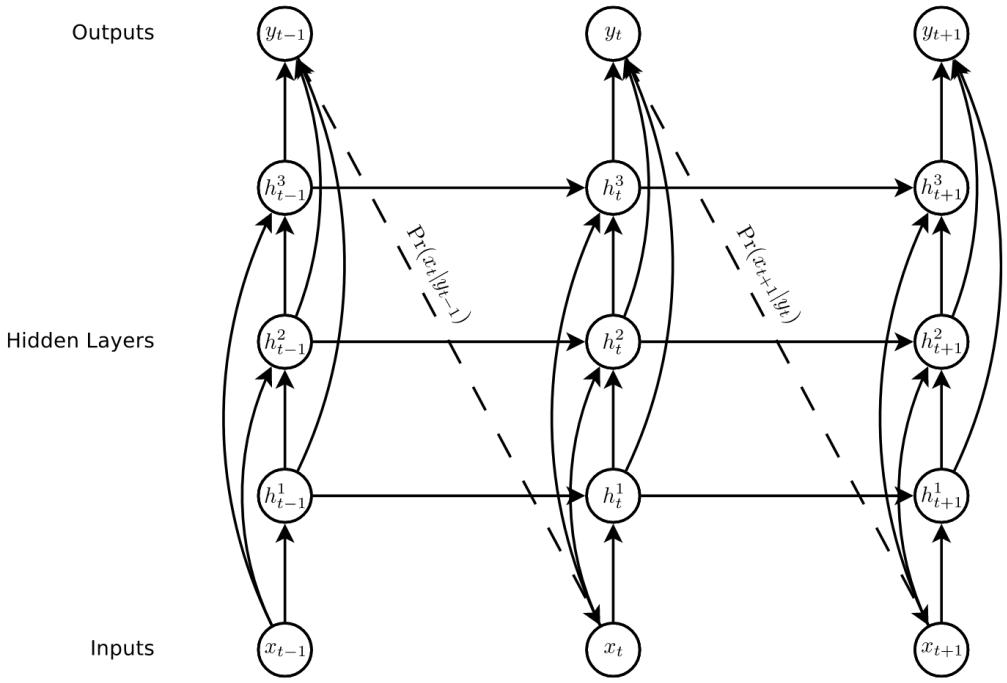
2.4 Recurrent Neural Networks

Despite the versatility of feed-forward networks, they are not optimized for problems with long-term sequenced inputs. In this regard, recurrent neural networks (RNN) arise as an alternative as they can scale to longer input sequences that would be practical for convolutional neural networks without sequence-based specialization (GOODFELLOW; BENGIO; COURVILLE, 2016). RNNs are a type of feed-forward neural network with feedback loops and internal states, allowing the information to persist between values on the input sequence, i.e., the network can keep track of history about the past elements that were presented on the input. The ability to retain past information is quintessential in applications where there is a temporal dependency between inputs, which justifies the excellence of these networks in applications like speech recognition, language translation, image captioning, video processing, and others (SAK; SENIOR; BEAUFAYS, 2014).

The outputs of the hidden units in a given RNN at different discrete time steps can be seen as if they were outputs of different neurons in a DNN (LECUN; BENGIO; HINTON, 2015). In fact, these networks were the inspiration for the spatial convolution networks (DONAHUE et al., 2017).

Figure 2.10 illustrates a deep recurrent neural network where the circles are the network layers, the solid lines are the weighted neural connections, and the dashed lines are the predictions. The *y*-axis indicates the network structure (layers), while the *x*-axis is the unrolled version of the network for multiple values of the input sequence. In these networks, an input vector $\mathbf{x}$ containing a sequence of T values is passed to a stack of N hidden layers that have recurrent connections to compute both the *hidden vector* se-

Figure 2.10: Deep recurrent neural network example



Source: (GRAVES, 2013)

quences $\mathbf{h}$ and output prediction $\mathbf{y}$. As stated in Graves (2013, p. 3),

> The network is 'deep' in both space and time, in the sense that every piece of information passing either vertically or horizontally through the computation graph will be acted on by multiple successive weight matrices and nonlinearities.

These traditional recurrent neural network, also known as vanilla RNN, are dynamic models which map the inputs to hidden states and vice-versa according to (2.10) and (2.11). The hidden state h_t is computed from a linear combination between the current input ($W_{xh}x_t$) and the hidden state value from the previous time step. This combination is passed to a non-linear function g, which is typically implemented as a sigmoid or hyperbolic tangent. Conversely, the output prediction y at time step t is computed solely in terms of the current hidden state.

$$h_t = g\left(W_{xh}x_t + W_{hh}h_{t-1} + b_h\right) \tag{2.10}$$

$$y_t = g\left(W_{hz}h_t + b_y\right) \tag{2.11}$$

In RNNs, the traditional backpropagation algorithm for weight update is insufficient to consider the influence of past inputs and states on the current input. Hence,

these networks rely on an extended version named backpropagation through time (BPTT) proposed by Zipser and Williams (1995), which unrolls the recurrent network so it can be seen as a feed-forward implementation, enabling the error measurement at each time step. However, BPTT does not scale well in vanilla RNN networks. As the number of time steps increases, the error gradient might vanish or explode, making it difficult for the network effectively learn any mapping function (HOCHREITER; SCHMIDHUBER, 1997). This situation is known as the *vanishing gradient* problem, and it is not limited to recurrent networks.

2.4.1 Long-Short Term Memory

An alternative to vanilla RNNs are the long short-term memory (LSTM) networks proposed by Hochreiter and Schmidhuber (1997), which were designed to overcome the gradient issues on the BPTT algorithm. LSTM-based networks support long input sequences, and they control the error gradient enforcing a constant error flow through the internal state of each LSTM unit. According to Gers, Schmidhuber and Cummins (2000, p. 2452),

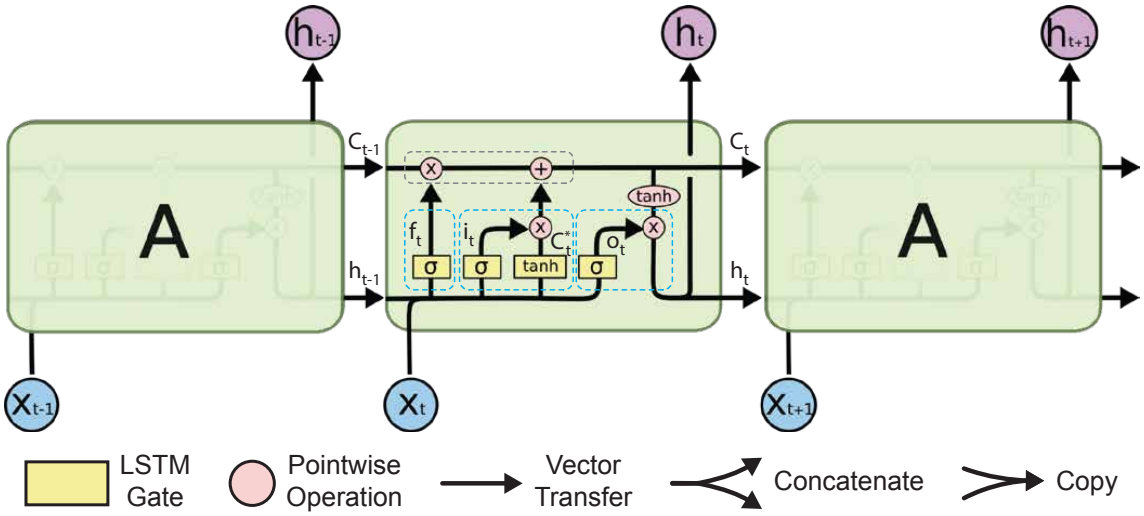
> The basic unit in the hidden layer of an LSTM network is the memory block, which contains one or more memory cells and a pair of adaptive, multiplicative gating units that gate input and output to all cells in the block. Each memory cell has at its core a recurrently self-connected linear unit called the constant error carousel (CEC), whose activation we call the cell state. The CECs solve the vanishing error problem: in the absence of new input or error signals to the cell, the CEC's local error backflow remains constant, neither growing nor decaying.

A compelling improvement to the original LSTM implementation was added by (GERS; SCHMIDHUBER; CUMMINS, 2000), where they introduced the *forget gate*. This gate is a hidden unit that controls the weight of this self-loop, effectively controlling when the memory blocks should be reset as their contents are no longer up-to-date. This version has been adopted as the standard LSTM implementation. There are other LSTM variations like the bi-directional LSTM and the gated recurrent unit (GRU), but they are beyond the scope of this work.

Fundamentally, LSTM networks are composed of LSTM cells which have an internal recurrence, also known as a *self-loop*, besides the outer recurrence inherent to RNNs (GOODFELLOW; BENGIO; COURVILLE, 2016). Each cell is composed of gates that control the information flow, leading to a higher number of parameters to be trained. These gates can be seen as recurrent network layers within the cell. Nonetheless, LSTM

cells still have the same input (sequence), and output (hidden state) interface as a regular RNN.

Figure 2.11: Internal LSTM unit implementation



Source: Adapted from (OLAH, 2015)

Figure 2.11 illustrates the implementation of an LSTM cell. The key element in these cells is the top line, which indicates the cell state (C_t): a memory block which keeps track of the influence of previous time steps. The cell state is updated according to the three control gates' outputs – forget, input, and output – which are delimited with a dashed blue line sequentially from left to right. All gates receive the concatenation of the previous hidden state (h_{t-1}) and the current input sequence (x_t). The dashed gray line highlights how the gates contribute to the cell state update mechanism.

The *forget gate* (f_t, shown in Fig. 2.11) controls how much the previous cell state (C_{t-1}) should be considered on the updated state. It applies a sigmoid function to the linear combination of x_t and h_{t-1}, constraining the output to the interval $(0, 1)$. If $f_t = 0$, the previous cell state is completely forgotten, else it carries some influence on the updated state. Mathematically, f_t is defined according to (2.12) where W_f is the weight set for the forget gate and b_f is the associated bias.

$$f_t = \sigma(W_f \cdot [h_{t-1}, x_t] + b_f) \tag{2.12}$$

The *input gate* (i_t and C_t^*, shown in Fig. 2.11) determines how much the current inputs (x_t, h_{t-1}) influence the cell state, i.e., it decides how relevant the input information is. That is achieved with two non-linear gates: i_t establishes the relevance of the inputs,

and C_t^* modulates the inputs, creating a vector of candidate values that could be added to the current state. In that sense, i_t is similar to f_t, so it employs a sigmoid function as the non-linearity while C_t^* employs a hyperbolic tangent (tanh) function to constrain these inputs to the interval $(-1, 1)$. Mathematically, these functions are defined according (2.13) and (2.14) where the pairs (W_i, b_i) and (W_c, b_c) are the weight sets and bias for i_t and C_t^*, respectively.

$$i_t = \sigma(W_i \cdot [h_{t-1}, x_t] + b_i) \tag{2.13}$$

$$C_t^* = \tanh(W_c \cdot [h_{t-1}, x_t] + b_c) \tag{2.14}$$

All the equations so far computed the gated values to update the old cell state. First, C_{t-1} is multiplied by f_t to determine how much influence the previous state has over the updated state. Then, this value is added to the modulated current candidate values computed on the input gate. These modulated values are obtained multiplying i_t by C_t^*. Formally, the updated cell state C_t is given by the following equation:

$$C_t = f_t \times C_{t-1} + i_t \times C_t^* \tag{2.15}$$

Likewise, the *output gate* (o_t, shown in Fig. 2.11) determines how much the updated cell state must be transferred to the hidden state. That is achieved using a sigmoid-tanh modulation similar to the input gate. In this case, the sigmoid function is applied to the input values (x_t, h_{t-1}) to decide what part of the cell state should pass to the output while the tanh function constraints the output to the $(-1, 1)$ interval. Note that the addition part of (2.15) may lead to a cell state value whose magnitude is greater than one; hence, it must be squashed to avoid the vanishing gradient problem. The formal definition of the output gate is given by (2.16) where W_o and b_o are the weight set and bias, respectively:

$$o_t = \sigma(W_o \cdot [h_{t-1}, x_t]) \tag{2.16}$$

$$h_t = o_t \times \tanh(C_t) \tag{2.17}$$

LSTM-based networks proved to be an excellent alternative to vanilla RNNs as they learn more easily the long-term dependencies (GOODFELLOW; BENGIO; COURVILLE,

2016). For instance, these networks achieved outstanding results in large-scale speech recognition (GRAVES; JAITLY, 2014) and language translation (SUTSKEVER; VINYALS; LE, 2014), whereas CNN-based networks are not suitable for these tasks. These networks have also been used in activity recognition in video files, image captioning (scene recognition), and video descriptions, among others (DONAHUE et al., 2017).

2.5 Training frameworks

The success of neural networks led to the development of several tools, libraries, and frameworks that help developers to build their models, so they do not need to implement everything from scratch. These tools offer all the mechanisms required for training and inference, like automatic gradient computation, layer instantiation, data analysis, and so on. They also embed optimized algorithms that can take great advantages of specific features in CPUs and GPUs (MOONS; BANKMAN; VERHELST, 2019). A non-exhaustive list of the most popular frameworks is listed below:

- **Caffe:** one of the first frameworks for neural networks ever deployed (JIA et al., 2014). It started as an academic project and grew into a large project used by many users. The network definition and deployment is not as straightforward as other frameworks.
- **Tensorflow:** it is a framework developed and supported by Google (ABADI et al., 2015). It provides several optimized functions and operations to create computational graphs for deep learning applications. Most of the time, it is used with the Keras wrapper, which provides state-of-the-art data manipulation and neural network implementations (CHOLLET et al., 2015). It is relatively popular due to its vast user base and well-written documentation, although its network graphs are static and precompiled, posing serious debugging challenges for the developer.
- **Pytorch:** this framework is maintained by Facebook (PASZKE et al., 2019), and it was initially based on the Torch implementation. It relies on dynamic graphs for on-the-fly compilation, making it easier to debug despite the speed penalty associated with it.
- **Theano:** it is similar to Tensorflow as it uses static graphs for network computation (THEANO DEVELOPMENT TEAM, 2016). Generally, it is used with the Lasagne (DIELEMAN et al., 2015) wrapper for more straightforward application

development as the latter features several functions and methods for neural network implementation.

2.6 Chapter Summary

This chapter aimed at presenting a comprehensive review and background on the most relevant concepts and methods on in the vast field of neural networks. Such concepts are necessary as the motivation for and basis over which the research described in the following Chapters is undertaken.

3 COPING WITH DEEP LEARNING COMPLEXITY ON EMBEDDED SYSTEMS

Deep recurrent neural networks have thousands, even millions, of parameters to execute the internal operations to obtain a prediction given an input. As networks grow deeper, the number of parameters and operations rapidly increases, demanding an ever-growing computational capability to process a network quickly and efficiently (MISRA; SAHA, 2010). However, edge devices hardly have the computing power nor the required energy in batteries to accomplish these tasks (VELASCO-MONTERO et al., 2018). Then, a hardware-software co-design approach is required to overcome the inherent complexity of these algorithms to extract the maximum performance attainable with the least amount of energy.

In this regard, there are three fundamental characteristics in deep learning that can be explored to optimize the algorithm execution and the underlying hardware. First, the unique characteristic of dataflow in deep neural networks foster data reuse and parallel computation. Second, efficient networks usually present a high degree of sparsity, i.e., most of the weights tend to be equal to or near zero, which can help to skip some operations to save time and energy. Finally, the intrinsic probabilistic nature of neural networks and the iterative learning process make these networks more robust to approximations, enabling the usage of data representations with a smaller number of bits and suitable for execution in tailored platforms based on approximate computing techniques. Most of these techniques require hardware support to be implemented efficiently.

Hence, this chapter reviews the most common co-design techniques that explore these three characteristics of neural networks. Then, it provides a brief overview of general and custom computational platforms for algorithm execution.

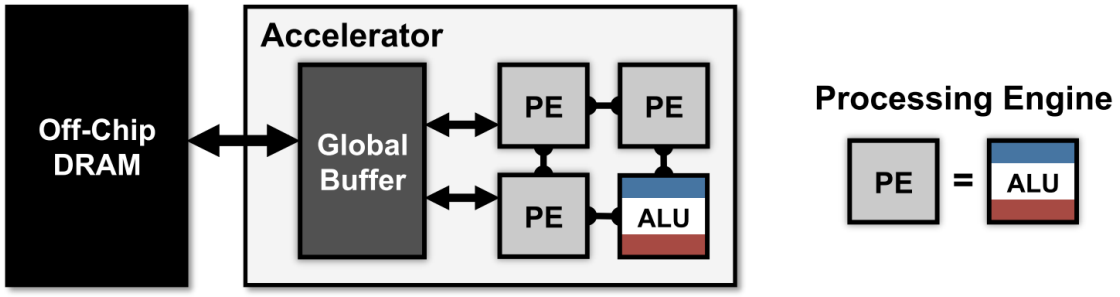
3.1 Exploiting network structure

Most deep neural networks rely on multiply-and-accumulate (MAC) operations within their convolutional and dense layers. Fundamentally, these operations can be easily parallelized with the additional benefit of reusing data for minimal data movement. Hence, system designers explore high-parallel compute diagrams, which include temporal and spatial architectures (CHEN; EMER; SZE, 2017). General-purpose hardware like CPUs and GPUs usually embed some type of temporal parallelization like single instruction, multiple data (SIMD) or single instruction, multiple thread (SIMT) techniques for

increase performance. Nonetheless, the parallelization degree attained in these platforms – in the order of hundreds to few thousand MAC units – is far from those presented in custom platforms like the Google TPU, which can compute 65.536 (256×256) MACs in a single clock cycle (JOUPPI et al., 2017).

In that sense, most custom hardware accelerators adopt the flow presented in Figure 3.1, aiming for as many simultaneous operations as possible with minimal data movement combined to an architecture flexible enough that it is not fixed to a single network structure. These architectures rely on an array of processing elements (PEs) where the operations are executed along with multiple memory levels for fast and energy-efficient accesses. Nevertheless, some deep neural networks may require specific accelerator architectures to cope with the application needs and system constraints like power dissipation and circuit area, leading to a trade-off between flexibility and execution efficiency at the circuit level.

Figure 3.1: Baseline DNN hardware accelerator



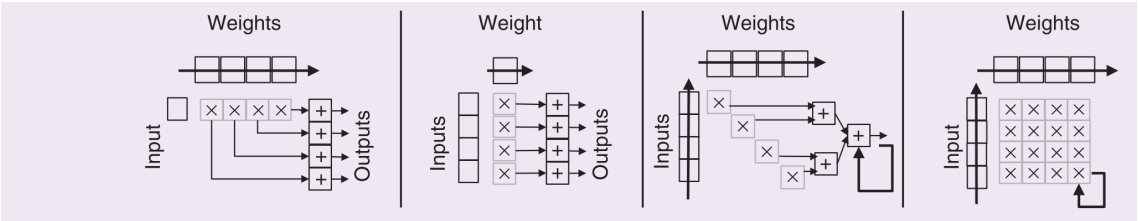
Source: (SZE et al., 2017a)

Custom accelerators are built to explore the temporal and spatial data locality inherent to deep neural networks to minimize data movements. For instance, the weights of a given CNN filter will be used over the entire input feature map to compute the outputs. Likewise, a given patch of the input feature map will be used by several filters in a given CNN layer. In both cases, memory access becomes a bottleneck when the parallelization is increased as a single MAC operation requires four memory accesses: input, weight and partial sum reads, and result write. Considering AlexNet, a popular CNN network, it would require near 3 billion DRAM accesses to compute a single output, severely penalizing power efficiency and system throughput (SZE et al., 2017b).

Hence, the first approach to minimize data movement relies on the adoption of a data reuse dataflow. Figure 3.2 summarizes three baseline topologies for data reuse, comparing the memory bandwidth requirements and the associated implementation. In

the *input stationary* approach, the same input data is multiplied by all filters for a given layer, i.e., the input is loaded only once and reused as much as possible before the next input. Note that this strategy allows the computation on multiple filters simultaneously, i.e., given an architecture implementation with multiple MAC units, each unit would have a different weight set, leading to a *weight parallel* implementation. Although this approach requires a very low input bandwidth, it severely penalizes the weight memory access, which must be loaded every cycle. Further, it prevents the MAC accumulation across different clock cycles, requiring the partial result to be stored in the memory to be fetched again later, impacting the output memory bandwidth.

Figure 3.2: Overview of data reuse topologies



	Input Stationary (Weight Parallel)	Weight Stationary (Input Parallel)	Output Stationary	Hybrids
Input BW	Low	High	High	Medium
Weight BW	High	Low	High	Medium
Output BW	High	High	Low	Medium

Source: (VERHELST; MOONS, 2017)

A similar approach is the *weight stationary*, where a given weight is loaded once and multiplied by the entire input feature map before the next weight load. When multiple compute units are available, they are configured for an *input parallel* scheme to maximize the data reuse. It takes a considerable toll on the input memory bandwidth since this strategy requires a new input every cycle. Like the previous approach, the MAC accumulation cannot span across clock cycles with an increased output memory bandwidth.

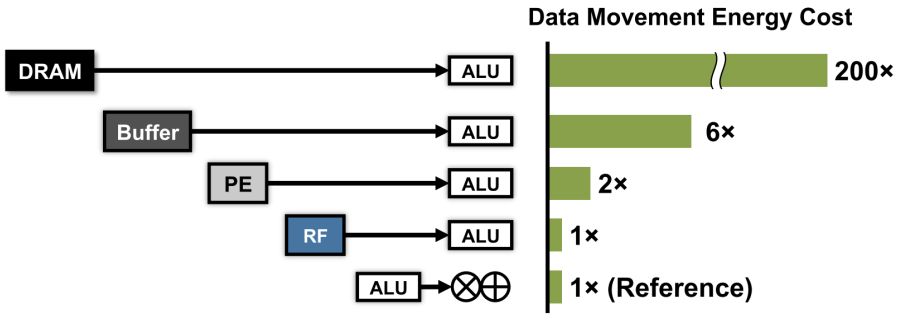
The third approach is the *output stationary*, which aims to minimize the output memory bandwidth by accumulating the partial results across multiple cycles and accessing this memory only after the final result is ready on the accumulator. Nevertheless, this approach requires loading both new weights and inputs every cycle, leading to high bandwidth requirement from these memories.

A balanced implementation is achieved with the *hybrid* approach, which is a combination of the previous three strategies. It requires a 2-dimensional MAC array where multiple input and multiple weights are loaded every cycle, and their multiplication is lo-

cally stored to be combined with previous partial sums. A comprehensive demonstration of this strategy is presented in Moons, Bankman and Verhelst (2019). A variant of this approach is known as *row stationary*, although the partial sums are transmitted across the PE units (CHEN et al., 2017). An energy comparison of reuse strategies is presented Sze et al. (2017b) considering the AlexNet network model where the row stationary approach achieves energy savings up to $2.5\times$ when compared to other strategies.

A complementary strategy relies on reducing the energy cost on data movement by introducing a multi-level memory hierarchy where each level has a different energy cost (SZE et al., 2017a). Local data movement, i.e., within the processing elements, is the cheapest movement operation in terms of energy while accessing the system RAM requires orders of magnitude higher energy consumption. Figure 3.3 illustrates the typical memory hierarchy division for these neural network accelerators. This strategy is only possible due to the temporal data locality in deep neural networks (MOONS; BANKMAN; VERHELST, 2019).

Figure 3.3: Energy cost of data movement at different levels



Source: (SZE et al., 2017a)

Another advantage of this approach lies in the possibility of tailoring the memory hierarchy and capacity for the target application. Optimizing the memory capacity and communication interface has a co-dependent link with the parallelization level adopted for the computing elements (MOONS; BANKMAN; VERHELST, 2019). This join optimization leads to a hardware system with minimal data movement, maximum data reuse, and increased energy efficiency. An extreme case is the systolic architectures like the Google TPU, where near all the data movement is kept within the PE array. In this case, the weights are preloaded on the functional units, and new input data moves from left to right on the PE array while the partial sums are accumulated on the bottom. Further details of this architecture can be found in Jouppi et al. (2017).

A promising strategy for minimal energy dissipation on data movement is the *in-*

memory computing, also known as *processing-in-memory* (PIM), where all computations are integrated into the memory itself. This approach is more energy-efficient than traditional digital designs with memory hierarchy as the data movement in embedded SRAM memories is extremely high (BISWAS; CHANDRAKASAN, 2018). Comprehensive reviews addressing the uniqueness of this approach have been published in (RAJENDRAN; ALIBART, 2016; DENG et al., 2020; KRESTINSKAYA; JAMES; CHUA, 2020).

3.2 Exploiting network reliability

First works in deep neural networks only focused on model accuracy, with no attention to algorithm complexity or hardware implementation challenges. However, DNNs are inherently robust to perturbations in both weights and activations, especially due to their non-linearities (VANHOUCKE; SENIOR; MAO, 2011).

Data representation is the first step towards efficient implementations of these networks. During training, the backpropagation algorithm often uses some normalization to avoid overfitting. Consequently, the weight and activation values fall within a limited range for each layer. Traditionally, all operations were performed on 32-bit floating-point since it available in nearly all processing hardware platforms, and most algebraic libraries are optimized for this representation.

Reducing precision is beneficial in several aspects. First, quantized models require less memory space and, consequently, less communication bandwidth. Second, this approach is hardware-friendly since it is much simpler to implement fixed-point or even an integer arithmetic unit when compared to the floating-point solutions. Third, hardware like GPUs with support for low-precision arithmetic can increase parallelism to improve energy efficiency.

Several works showed that neural networks with reduced precision could achieve nearly the same accuracy of those implemented using floating-point operations. Courbariaux, Bengio and David (2014) performed an extensive analysis of several data representation approaches, including floating-point, fixed-point, and dynamic fixed-point. Results show that reducing from a 32-bit floating-point implementation down to 12-bit dynamic fixed-point has a marginal increase on the error rate, indicating that neural networks can indeed operate at a lower precision.

An extreme quantization approach was proposed by Courbariaux, Bengio and David (2015), where the authors constrained all the weights to either $+1$ or -1 while

keeping the backpropagation gradients with sufficient precision to update the network parameters correctly. In some cases, the weight constraint regularizes the network, increasing its accuracy. Similar approaches were followed by Zhou et al. (2016) and Hubara et al. (2016).

A generalized quantization approach is presented in Moons et al. (2017). In this work, the authors perform a comprehensive analysis of the trade-off between accuracy and energy efficiency for multiple bit-widths. For a custom network, the optimal data bit-width ranges from 1 to 4 bits, depending on the accuracy requirements.

The reliability of neural networks also tolerates non-deterministic errors caused by computation executed on the analog domain and in digital circuits operating in the near-threshold region (MOONS; BANKMAN; VERHELST, 2019). Reducing the operating voltage leads to memory failures and increased delay, which may cause timing violations. Nonetheless, DNNs can absorb part of stochastically induced errors when combined with additional hardware support to monitor the circuit fault rate (LIN; ZHANG; SHANBHAG, 2016).

3.3 Exploiting network sparsity

All state-of-the-art neural networks require millions of parameters to describe an accurate model for classification tasks. The majority of these parameters reside on the classifier, composed of fully-connected layers. However, some connections in a neural network have minimal impact on model accuracy. For instance, an AlexNet model requires more than 200 MB of storage while a VGG-16 network occupies more than 500 MB to store the network parameters (HAN; MAO; DALLY, 2015). Combining the inherent sparsity of neural networks with limited representation bit-width results in many parameters tied to zero, which can be exploited for optimal energy efficiency.

In Hinton et al. (2012), the authors proposed a network where the neural connections within the classifier were probabilistically pruned, i.e., the weights were set to zero. For the CIFAR-10 classification dataset, the overall network error rate improved to 15.6%, while the baseline model achieved an error equal to 16.6%, illustrating how neural networks can be severely affected by overfitting.

Hardware-wise, all MAC operations with either input or weights tied to zero can be skipped entirely, avoiding spending energy to fetch and store data from memory. Further, the PE units can be implemented with a data-gating approach to avoid changing its

internal state when either input or weight is zero (VERHELST; MOONS, 2017). The sparse representation also enables off-chip data compression to minimize the energy cost of data movements between the hardware accelerator and the system memory (HAN; MAO; DALLY, 2015). Simple compression schemes like Huffman require a small additional circuitry with significant power savings.

Additional optimization steps involve iterative parameter pruning using energy consumption models to maximize pruning efficacy, eliminating up to 90% of model parameters (YANG; CHEN; SZE, 2017). Some approaches also combine quantization, pruning, and data compression with even tailored datapath for inference operations directly over the compressed data, removing the deep of a decompression step on the system (HAN et al., 2016). A comprehensive review of network pruning is presented in Sze et al. (2017b).

3.4 Computational Platforms

The works of Steinkrau, Simard and Buck (2005) and Chellapilla, Puri and Simard (2006) found that GPGPUs could be used to train neural networks. As these cards aim for the parallel processing of several multiplication and addition operations, they were a perfect match due to the nature of these algorithms, especially for convolutional neural networks.

Since 2011 with AlexNet, the networks have become deeper, reaching up to hundreds of tasks, demanding billions of operations to process a single input. Further, training datasets are growing aggressively like, for instance, the ImageNet data that contains 1.2 million images. One of the first image datasets publicly available was MNIST, which included images of handwritten digits, and it has only 60000 training images.

3.4.1 Central Processing Unit (CPU)

Modern processors embed several instruction sets tailored to efficiently operate on multiple data at once, aiming to improve the system throughput. Superscalar processors support SIMD instructions that can perform the same operation over multiple data in an efficient way (PATTERSON; HENNESSY, 2013). This is particularly interesting in applications like convolutional neural networks since they require several multiplication

and addition operations to be executed over an entire feature map.

Current machine learning frameworks have embedded support for SIMD instructions as well as interprocess communications libraries to speed the training/inference process (RYBALKIN et al., 2017). Additionally, several BLAS (Basic Linear Algebra Subprograms) libraries are available to improve mathematical operations – like matrix multiplication – by efficiently exploiting these instructions (VANHOUCKE; SENIOR; MAO, 2011). Most machine learning frameworks embed native support for these instructions.

Even with optimized instruction sets and assembly-optimized matrix multiplications, the performance is not optimal in CPU-based implementations as they have to divide processing timing with other processes. Further, the lack of an optimized memory hierarchy scheme for this type of application severely affects the computational speed due to the frequent access to the RAM to read and write weights and activations.

3.4.2 Graphic Processing Unit (GPU)

Due to its inherent data-centric and dataflow-oriented structure, GPUs are efficient off-the-shelf solutions for both training and inference phases of neural networks. At their inner root, neural networks and video/image processing revolve around matrix multiplications. Thus they share the same type of computation. The relative low-cost, high availability and processing power are the reasons behind the GPUs' dominance in the neural network processing field.

Similarly to the BLAS libraries offered for general-purpose CPUs, libraries like cuBLAS (NVIDIA, 2008) and cuDNN (CHETLUR et al., 2014) offers significant performance increase on GPUs compared to a standard execution on these platforms. For instance, cuDNN offer a training time speedup about $1.36\times$ for a convolutional neural network implemented on the Caffe framework (CHETLUR et al., 2014).

In Li et al. (2016), the authors explored a plethora of CPU and GPU combinations aiming for faster training and inference times and energy efficiency. The authors considered the ConvNet benchmark to obtain a fair comparison between results. Despite the higher instant power, GPUs perform better regarding throughput and energy efficiency. For instance, a Titan X GPU required around 0.25J per image for a batch size 32 while a CPU-based implementation consumed around 4J per image – i.e. $16\times$ more energy – under the same conditions.

3.4.3 Hardware Accelerators

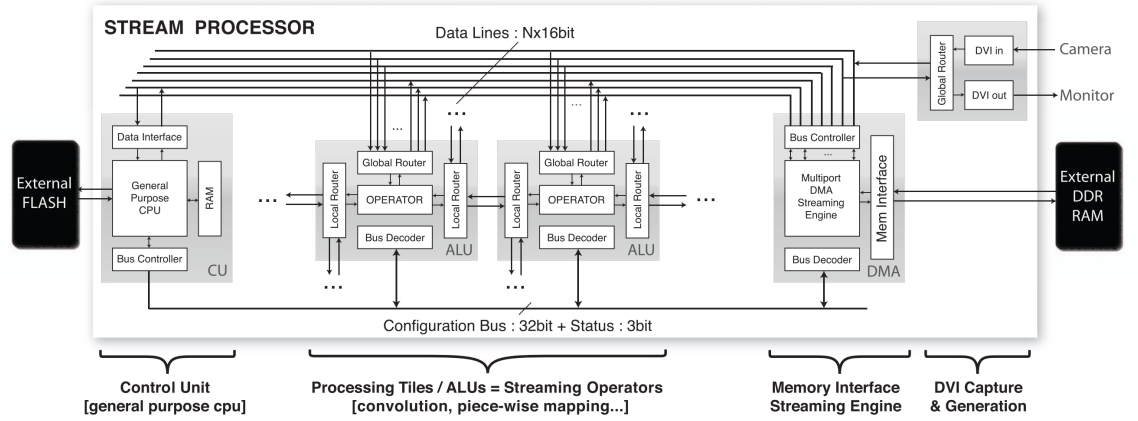
Although GPUs offer an enormous advantage over current CPUs for neural networks, they still consume a considerable amount of energy. Since the current trend is to move the inference phase to the node that usually has a power source limitation, it is necessary to develop power-efficient software and hardware solutions. For mobile systems, CPUs and GPUs are far from being the ideal platforms due to the performance and power dissipation limitations. Hence, the solution resides on custom hardware accelerators implemented in either Field Programmable Gate Arrays (FPGA) or Application-Specific Integrated Circuits (ASIC).

First, hardware implementations targeted custom architectures for specific neural networks with two main objectives: first, huge speedup when compared to CPU-based implementations and, second, overwhelming improvements concerning energy-efficiency when compared to GPU-based applications. In fact, Nurvitadhi et al. (2017) indicates that these custom hardware platforms may outsource CPU and GPU as computing platforms for neural network accelerators since they can better exploit newer DNN algorithms that integrate reduced precision operations, custom memory access schemes, zero-skip computations and so on.

The first neural network hardware accelerator was proposed by Farabet et al. (2009), targeting an FPGA implementation. The proposed design employed a PowerPC-based soft-core implementation to control the Vector Arithmetic and Logic Unit (VALU) that performed all the CNN-specific operations like convolution, pooling, and so on. An evolution of this hardware was proposed in Farabet et al. (2010), and it was able to process input images at real-time consuming near around 15W, according to the network specification. Figure 3.4 illustrates the architecture of the proposed stream-based processor. Each processing tile is directly connected to its processing neighbors, and all system modules are connected through multiple 16-bit data channels. The configuration bus is reserved for setting the parameters of each processing tile according to the convolutional network parameters.

Since then, several works aimed for custom frameworks and design flows to automatize the mapping procedure from the algorithm level to the hardware level. Wei et al. (2017) proposed an automatic synthesis of CNN accelerators based on systolic arrays. The authors follow a high-level synthesis (HLS) approach, converting C code directly into FPGA bitstream. In Guo et al. (2018), the authors propose a hardware accelerator as

Figure 3.4: ConvNet Stream Processor architecture



Source: (FARABET et al., 2010)

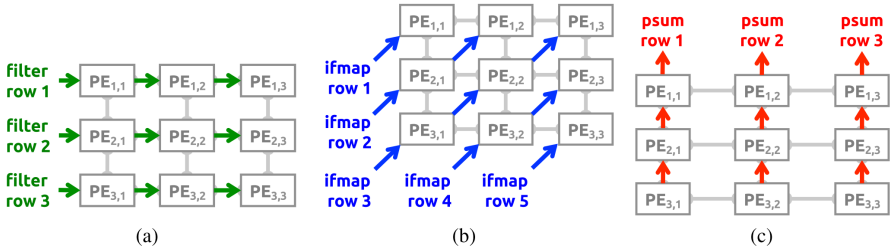
well as a complete mapping flow for CNNs into FPGA devices. It includes data quantization strategies, run-time configurable hardware for multiple network configuration, and a compiler to map the CNN model onto the hardware architecture.

DianNao was one of the earliest ASIC hardware accelerators for convolutional neural networks, and it was proposed in (CHEN et al., 2014) considering a 16-bit datapath. The accelerator features a neural functional unit (NFU), which is responsible for performing the synapse-related computations. The NFU is controlled with a custom instruction set, enabling the execution of generic neural networks. This accelerator outperformed a conventional SIMD core in both throughput and energy consumption. Considering only convolutional and classifier layers, the proposed design is, on average, 117.87$\times$ faster and consumes 21$\times$ less than the baseline core. A family of accelerators was derived from this implementation leading to other task-specific designs. ShiDianNao (DU et al., 2015) was conceived for low-power and consumes $60\times$ less than DianNao. The DaDianNao implementation focused on high-throughput applications and achieved a top performance of 5585 GOPS (CHEN et al., 2016).

Chen, Emer and Sze (2016) proposed the Eyeriss accelerator, and it maximizes the memory reuse within the lower memory levels, thus reducing the energy consumption required by the data movement on the upper levels. The energy efficiency attained by this circuit is a consequence of three techniques. First, it breaks a multi-dimensional convolution into several 1D convolutions, hence its row stationary characteristic. The dataflow uses a two-step primitive mapping that occurs before run-time to manage these primitives. Finally, it employs an optimized data handling approach to maximize the usage of the storage hierarchy.

Figure 3.5 illustrates how the data is reused on the Eyeriss processor considering a 3×3 PE set. Considering a filter kernel of size 3×3, each filter row is reused across the PEs, i.e., all PEs on the same row will store the three weights of the filter row (Figure 3.5a). Each row of the input feature map is propagated across the PEs diagonally (Figure 3.5b), and, finally, the partial sums are accumulated across the PEs vertically (Figure 3.5c). The combination of these three strategies enables the full computation of, at least, three rows.

Figure 3.5: Row Stationary dataflow



Source: (CHEN; EMER; SZE, 2016)

Some applications do not always require high precision or even exact computing to present satisfactory results. In compute-intensive applications like ML algorithms, there is an excellent opportunity to improve hardware accelerators' energy efficiency. The first work to explore approximations at the hardware-level for neural networks is found in Du et al. (2014). The authors show that using inexact logic the energy savings are up to 62.49% with a mean square error increase from 0.14 to 0.20.

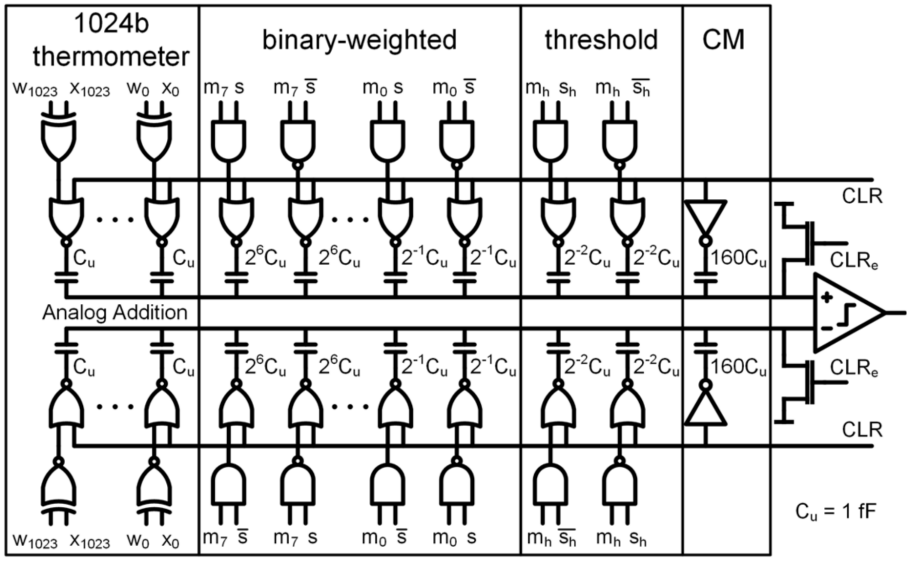
The ApproxANN framework, proposed by Zhang et al. (2015), considers the approximation at both computation and memory access level for custom hardware accelerators. First, it performs an analysis to determine which neurons have a more substantial impact on the output and label them as critical neurons. Based on this importance-based neural map, the framework creates a rank indicating which neurons are most likely to be approximated. Assuming the hardware accelerator shown in Figure 3.3, there are three approximation mechanisms: a) processing elements may skip computation and memory accesses if the neuron being processed is not critical; b) reduced data precision through truncation and c) approximate hardware for arithmetic operations. The authors explored several approximation configurations to find a trade-off between application quality and energy savings. For instance, on the MNIST dataset, the approximate version enabled energy savings up to 35% with less than 0.5% on accuracy loss.

When considering extreme quantization approaches, Andri et al. (2016) intro-

duced a hardware accelerator for binary neural networks where all weights were constrained to -1 and $+1$. These values were mapped to 0 and 1 to simplify the hardware implementation. Adopting this design approach greatly simplifies the hardware as adders and multiplexers can replace all MAC units. The proposed circuit achieves an outstanding energy-efficiency peak of 61 TOPS/W for 65nm ASIC implementation with 0.6V supply voltage.

One of the earliest hardware accelerators to combine a dynamic voltage-frequency-accuracy scaling (DVFAS) approach was proposed in Moons and Verhelst (2017). This ConvNet processor has a 16×16 MAC array controlled by a dedicated ASIP, and its datapath can be configured at run-time to operate on from 1 to 16 bits at multiple frequencies. Its ASIP-based approach simplifies the usage of this accelerator since it is fully C-programmable. It also features data-gating to skip computation when either weights or activations are zero, which has a significant impact on the dynamic power. Considering the AlexNet network as the benchmark, this processor outperforms other state-of-the-art dedicated processors up to $5\times$ regarding energy-efficiency with a 35% higher frame rate.

Figure 3.6: Switched capacitor-based neuron



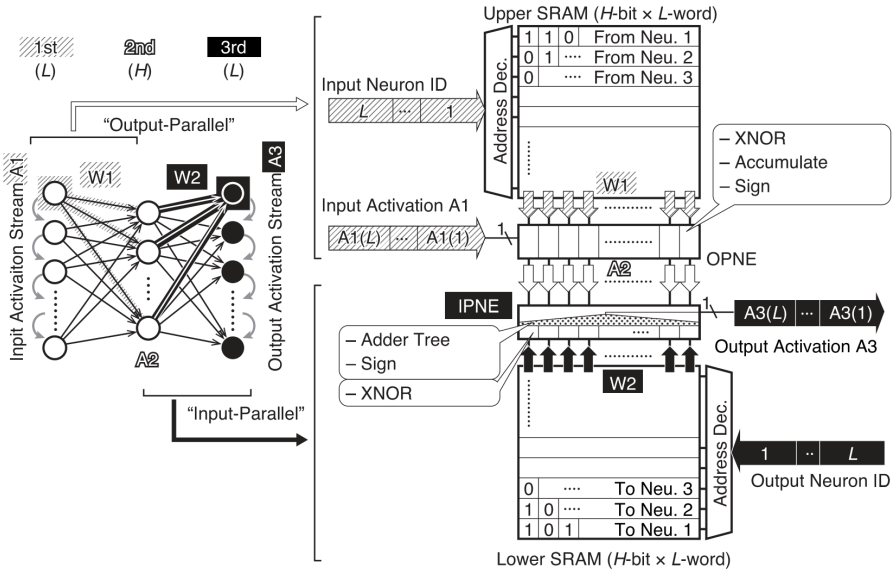
Source: (BANKMAN et al., 2018)

Using processing elements near memory and adopting mixed-signal design flows are trending approaches to the design of neural network accelerators due to the improved energy efficiency on these circuits. Bankman et al. (2018) constrained both weights and activations to -1 and $+1$. In this case, the entire multiplication datapath is resumed to several XNOR gates. The authors also employed a mixed-signal approach where the sum of neuron inputs is achieved through a switched capacitor bank, enabling the circuit

operation at low voltages around 0.6V – without a significant sacrifice on its performance. Due to its 1-bit datapath and analog-based adder, the proposed system consumes only 3.86μJ to classify an image using the MNIST dataset. Figure 3.6 shows the structure of the switched capacitor based neuron. The thermometer section applies the 256 2×2 filters to compute the neuron output, which is summed to a 9-bit bias. The threshold section implements the activation function.

Instead of focusing only on CNN-specific accelerator, Ando et al. (2018) proposed the BRein chip. This deep neural network accelerator can be configured to operate as either a fully-connected network or as a convolutional neural network. The circuit also supports both binary and ternary neural networks, enabling the use of higher accuracy applications. The fundamental concept in this circuit is the use of processor-in-memory modules that reduce the energy cost of data movement. Each PIM module can process three layers, and it uses two SRAM banks – upper and lower modules – with an arithmetic core between them to compute the neural activation, as seen in Figure 3.7. The silicon prototype supports network with up to 13 layers, and it has a peak energy efficiency of 2.3 TOPS/W.

Figure 3.7: PIM Module on the BRein Architecture



Source: (ANDO et al., 2018)

3.5 Chapter Summary

This Chapter reviewed key concepts on the hardware execution and hardware platforms that present different trade-offs for executing NN kernels – the essential and most important operations that are demanded by DNNs. Prior works geared toward hardware acceleration of NN were reviewed, as they introduced key strategies for guiding future research on this field. Table 3.1 presents a summary of the aforementioned optimization techniques and strategies as well as how they can applied to different platforms to cope with the inherent complexity of neural network models.

Table 3.1: Summary of Optimization Techniques and their Applications

Class	Strategy	Goal	Platform suitability	Implementation Complexity
Network Structure	Temporal parallelization	Explore SIMT and SIMD instructions on GPUs and CPUs	CPU, GPU	Low
	Multi-level memory hierarchy	Segment the memory accesses into hierarchical levels to reduce the memory access energy cost, specially in most frequent data	FPGA, ASIC	High
	Data reuse	Minimize data movement between memory hierarchy for faster processing and lower energy consumption	FPGA, ASIC	High
	In-memory Computing	Integrate the circuit logic to the memory itself, reducing the time and energy required to move data	ASIC	High
Network Reliability	Data quantization	Reduce the number of bits for data and parameters representation, reducing the risk of overfitting and improving memory usage and energy consumption	GPU*, FPGA, ASIC	Medium
	Binarization	Extreme data quantization scheme where data/parameters are represented with a single bit.	FPGA, ASIC	Medium
	Voltage scaling	Circuits may be set to operate to a lower power supply voltage for increase energy performance at the expense of some impact on the accuracy due to timing errors.	FPGA*, ASIC	High
Network Sparsity	Weight Pruning	Some weights on neural network can be removed (set to zero) and their computation can be skipped.	CPU*, GPU*, FPGA, ASIC	Low
	Data compression	Sparse models can be compressed to minimize the energy cost to move the data to/from the off-chip memory.	FPGA, ASIC	High

*The technique cannot be fully explored in this platform

4 ARITHMETIC KERNELS ON NEURAL NETWORKS

Recent advances in neural network architectures focused on improving the overall accuracy while limiting the model complexity in terms of parameters and computation. State-of-the-art networks like require hundreds of millions of MAC operations to infer the output for a single input. Combining this amount of operations with system-specific latency requirements in battery-constrained devices imposes a severe challenge for designers.

Convolutional layers require very little storage memory as the kernel coefficients are shared within a feature map. Conversely, these layers are responsible for most of the computation time for a given neural network. The work of Cong and Xiao (2014) shows that for the AlexNet CNN, which has five convolutional layers and three fully-connected layers for the classifier, the convolutional layers consume near 91% of the inference time while the classifier uses less than 1% of the time to perform its computation. This compute-intensive characteristic of the convolutional layers offers several optimization approaches concerning hardware acceleration in terms of both energy consumption and achievable throughput.

Nonetheless, LSTM layers have different computation requirements as they have fewer arithmetic operations with higher bit-width. The binarization process that is usually adopted in CNN layers cannot be fully explored in recurrent layers due to the dependence on previously accumulated states. In this case, multipliers and adders play an essential role, especially in terms of critical path and dynamic power dissipation.

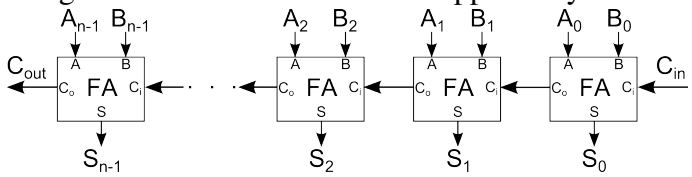
4.1 Two-operand Binary Adders

Adders are the basic building blocs of nearly all arithmetic circuits. Hence their optimization is quintessential for resource optimization (DESCHAMPS; BIOUL; SUTTER, 2006). This circuit can be employed in more complex architectures like multipliers and multi-operand adders.

4.1.1 Ripple Carry Adder (RCA)

The most straightforward approach for multi-bit addition is chaining full adders (FA) where the carry-out from the addition at weight i is transmitted to weight $i + 1$ as the carry-in. In this way, the carry ripples from the least significant bit (LSB) to the most significant bit (MSB), as illustrated in Figure 4.1

Figure 4.1: Structure of a n-bit ripple carry adder



Source: The Author

Due to its simplicity, no additional logic is required to compute the carry chain, leading to the smallest area possible for any given operand size. Assuming that each full adder has a critical path of two CMOS logic gates whereas each gate has a delay D, the critical path for a n-bit ripple carry adder is given by the carry chain whose total delay is given by:

$$T_{delay} = \overbrace{2D}^{C_{O_{n-1}}} + \overbrace{2D}^{C_{O_{n-2}}} + \cdots + \overbrace{2D}^{C_{O_0}} = n \times 2D \tag{4.1}$$

From this equation, it is clear that as the bit-width increases, the time taken to reach the result increases linearly because each bit depends upon the result of the previous bit. The required area to accommodate such an adder may be calculated in terms of logic gates, ignoring the differences in drive-loading capabilities, and others. Assuming an FA implementation with five logic gates, the total area for a n-bit RCA is given by:

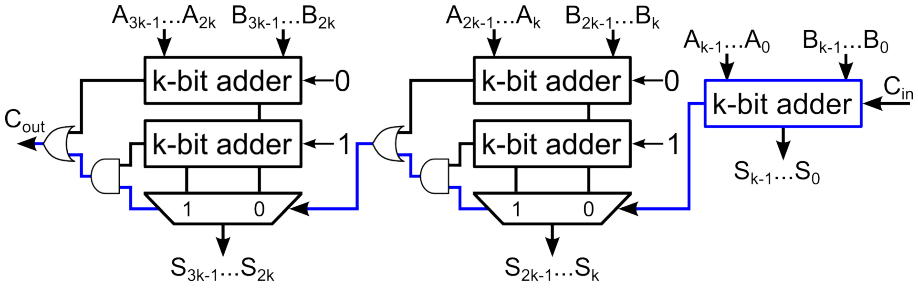
$$T_{area} = \overbrace{5A}^{FA_{n-1}} + \overbrace{5A}^{FA_{n-2}} + \cdots + \overbrace{5A}^{FA_0} = n \times 5A \tag{4.2}$$

4.1.2 Carry Select Adder

Based on the RCA structure, the carry select adder combines a block-based approach and the conditional sum principle to pre-compute slices k bits of the final result. Each adder block is duplicated, and each replica has the same inputs except for the carry-in. Then, a multiplexer selects which replica's output will be copied to the adder output

based on the carry-out of the last block. Assuming that all blocks have the same number of bits, all the results will be ready simultaneously, resulting in a critical path composed of a single k-bit adder and the following output selection circuits, as highlighted in blue in Figure 4.2.

Figure 4.2: Structure of a fixed-group size carry-select adder



Source: The Author

According to Tyagi (1993), the carry select adder has a critical path complexity proportional to $O(n^{1/2})$ when the block sizes are set to the optimal size. Further, using variable block sizes can improve performance when the gate delays are well characterized at the design time. In this case, the first group has a small ripple carry adder. In contrast, the subsequent groups may feature wider RCAs to match the output selection propagation time, effectively reducing the computation delay.

4.1.3 Carry Look-Ahead Adder

Despite its small area, the ripple carry adder does not scale well in timing as the operand sizes increases, due to the nature of its carry propagation path. Given that all input operands' bits are ready at a given time, the carry can be pre-computed to accelerate the propagation path based on the following properties:

1. When two bits at the same weight i are equal to one, a carry-out is generated regardless of the carry-in value. This is the *carry generate* function that is implemented as the logic AND between the input operands.
2. When only one of the inputs is equal to one, the carry-in will be propagated to the carry-out. This is the *carry propagate* function that is defined as the logic XOR between the input operands.

Adders that are implemented based on these functions belong to the carry look-ahead adder (CLA) family, and they are able to dramatically reduce the time to complete

a sum. Adopting G_i for the generate function for the i^{th} bit and P_i the propagate function for the i^{th} bit, Katz (1994, p.264) shows that "sum and carry-out can be expressed in terms of the carry generate and carry propagate functions":

$$S_i = \overbrace{A_i \oplus B_i}^{P_i} \oplus C_i = P_i \oplus C_i \tag{4.3}$$

$$\begin{aligned} C_{i+1} &= A_i B_i + A_i C_i + B_i C_i && \text{(4.4a)} \\ &= A_i B_i + C_i(A_i + B_i) && \text{(4.4b)} \\ &= A_i B_i + C_i(A_i \oplus B_i) && \text{(4.4c)} \\ &= G_i + C_i P_i && \text{(4.4d)} \end{aligned}$$

The logic transformation made from (4.4b) to (4.4c) is possible because the generate function will cover the results when both A_i and B_i inputs are equal to one, eliminating the possibility of two true inputs on the OR gate. These functions can then be arranged recursively to parallelize the evaluation of each input carry which must arrive at the full adder cells. Suppose a n-bit adder and assume that the first carry is given with the input, the carry-in for each full-adder cell can be computed as follows:

$$\begin{aligned} c_0 &= C_{in} && \text{(4.5a)} \\ C_1 &= G_0 + P_0 C_0 && \text{(4.5b)} \\ C_2 &= G_1 + P_1 C_1 = G_1 + P_1 G_0 + P_1 P_0 C_0 && \text{(4.5c)} \\ &\vdots \\ C_{n-1} &= G_{n-2} + P_{n-2} C_{n-2} = G_{n-2} + P_{n-2} G_{n-3} + P_{n-2} P_{n-3} C_{n-3} = \ldots && \text{(4.5d)} \end{aligned}$$

Despite the speed advantage due to the parallelization of the carry computation chain, the logic depth grows considerably for larger inputs, resulting in a huge die area to accommodate the circuit. According to Katz (1994), as the inputs widen, more carries need to be computed, requiring OR gates with an unfeasible number of inputs.

Hence, this baseline implementation of CLA is hardly used in modern circuits due to their area inefficiency. Nevertheless, the parallel prefix computation is a strategy that aims to mitigate the CLA computation issues since the carries "can be computed as a chain of prefix operations" (KNOWLES, 2001, p.278). All parallel prefix-based adders compose a new family of arithmetic circuits whose timing complexity is proportional to

$O(\log n)$.

4.1.3.1 Kogge-Stone Adder

The implementation of all parallel prefix adders rely on the operator "*o*" described in Kogge and Stone (1973), and is defined in (4.6) where (g, p) and $(\hat{g}, \hat{p})$ are the generate and propagate functions of bit i and bit $i-1$, respectively. The final carry propagate and generate values at the i^{th} position are given by G_i and P_i, respectively, which rely on the recursive concatenation of the operator *o* according to (4.6).

$$(g, p)o(\hat{g}, \hat{p}) = (g + (p\hat{g}), p\hat{p}) \tag{4.6}$$

$$(G_i, P_i) = \begin{cases} (g_1, p_1) & \text{if } i = 1 \\ (g_i, p_i)o(G_{i-1}, P_{i-1}) & \text{if } 2 \leq i \leq n \end{cases} \tag{4.7}$$

Due to the associativity of this operator, the function does not need to be calculated sequentially. This operator, henceforth named *black cell*, is mapped to a hardware circuit according to (4.8) and (4.9). Since the fundamental CLA equation has a linear recurrence and all their input values are well-defined as a consequence of the existent PG functions, they can be solved using the idea of recursive doubling introduced by Kogge and Stone (1973).

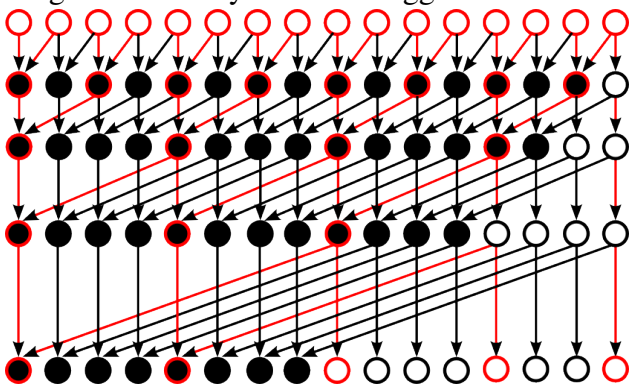
$$P_{out} = P_{in}\widehat{P_{in}} \tag{4.8}$$

$$G_{out} = G_{in} + (P_{in}\widehat{G_{in}}) \tag{4.9}$$

The Kogge-Stone adder relies on a carry computation tree composed of black and white cells, whereas the latter ones are just signal buffers to transmit the values between logic levels. As shown in Figure 4.3, this implementation aims to compute all the carries as soon as possible, keeping a constant fanout for both black and white cells. Nevertheless, this constraint usually leads to increased wiring capacitance, demanding the insertion of buffers to mitigate this problem (KNOWLES, 2001).

Given that this architecture features the maximum computation parallelism on the carry tree, it represents the fastest binary adder architecture with a critical path proportional to $O(\log(n))$. However, this maximum speed is only theoretical, and it is hardly achieved due to the massive hardware replication and the massive amount of interconnec-

Figure 4.3: Carry tree of a Kogge-Stone adder



Source: The Author

tions, leading to increased capacitance and wire length caused by routing congestion.

Some variants of the Kogge-Stone try to mitigate the drawbacks of the original implementation, and they mainly focus on two strategies. First, some architectures explore higher-radix processors that compute the propagate-generate functions considering more than two bits simultaneously. Another popular approach relies on computing fewer carries, which leads to a sparse tree with fewer components whose outputs are fed to other adders in a mixed architecture. The cells and connections highlighted in red in Figure 4.3 indicate the remaining components of a sparsity-4 Kogge-Stone adder, resulting in a smaller circuit at the expense of a longer critical path.

4.1.3.2 Brent-Kung Adder

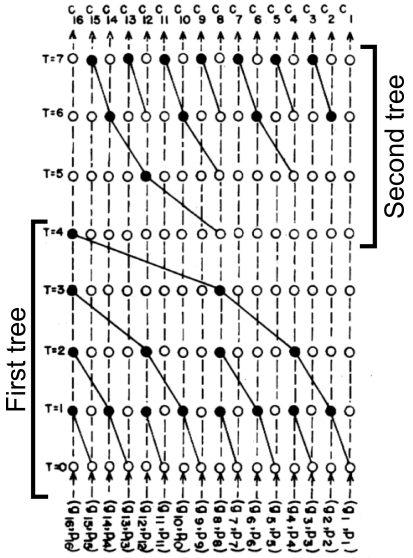
The Brent-Kung adder was proposed by Brent and Kung (1982), and it is an alternative to the Kogge-Stone implementation, which focuses on circuit regularity and less wiring congestion, that may improve the adder performance. The design relies on the same white and black cells previously introduced to compose the carry computation tree, which has two subtrees.

The first tree adopts a leaf-to-root approach like an inverted binary tree. In the first tree level, the black cells are directly connected to the inputs. Then, at each subsequent level, the black cells combine the pair of results computed on the previous level until the tree is completely generated up to the last bit. From the properties of the binary tree, it is straightforward to observe that all carries whose position is a power of two will have their final values computed.

The second tree is responsible for computing the remaining carries using a root-to-leaf approach. In this case, the tree root is located at the power of two nearest to

the center. Then, the tree follows the same algorithm as the first tree. To illustrate this operation, Figure 4.4 shows the complete carry tree for a 16-bit wide adder.

Figure 4.4: Carry tree of a Brent-Kung adder



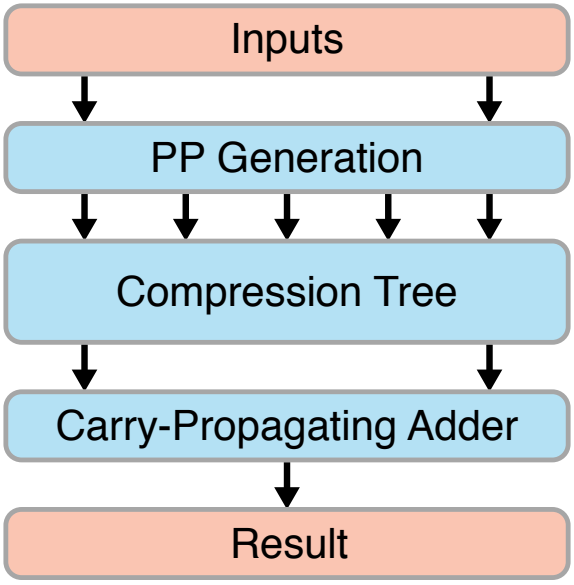
Source: Adapted from (BRENT; KUNG, 1982)

Two fundamental properties are found on Brent-Kung adders. First, the number of carry computation levels is equal to $2\log_2 n - 1$, proving that the critical path increases logarithmically with the input size. Further, each black cell's fanout is constant and equal to two, reducing the load requirements. Brent and Kung (1982, p.262) affirm that the area occupied by this design is quasi-linear, that is, it increases proportionally to $n \times \log n$.

4.2 Parallel Binary Multipliers

The hardware implementation of a multiplier can be divided into three blocks, according to Figure 4.5. The first block is the partial product generation algorithm. Then, as these algorithms usually generate more than two multi-bit signals, they require a compression tree based on the carry-save scheme to reduce the partial products to only two values. Since there are only two remaining values, the are recombined using a carry propagating adder.

Figure 4.5: General architecture for parallel binary multipliers



Source: The Author

4.2.1 Partial Product Generation Algorithms

Choosing the optimal partial product generation is quintessential as it has an enormous impact on how the compression tree is built and the hardware associated with it. Among the most common partial product generation algorithms is the Modified-Booth, optimized Baugh-Wooley array, and the Radix-2^m (COSTA; BAMPI; MONTEIRO, 2002) for signed multiplication. Although these algorithms are conceived for signed inputs, they can also compute unsigned operations is the input data is constrained.

4.2.1.1 Booth Multiplier

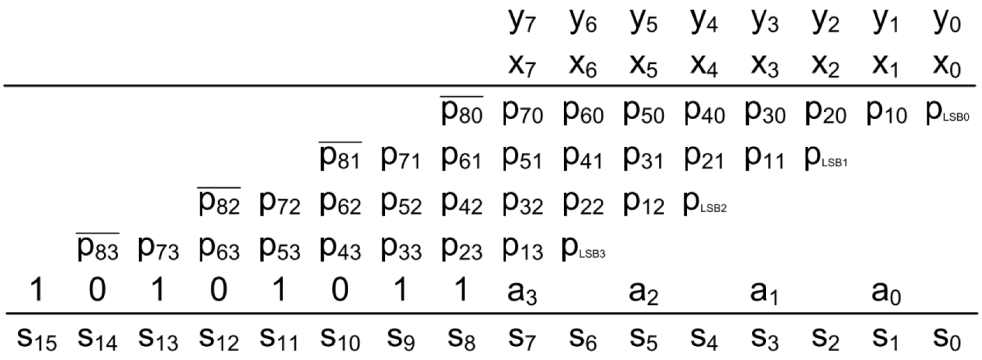
The first approach to compute signed multiplications was the Booth algorithm on which "binary numbers of either sign may be multiplied by a uniform process that is independent of any foreknowledge of the sign of these numbers" (BOOTH, 1951, p.1). The Booth multiplier assumed that both input were represented in two's complement. However, this algorithm is not suitable for hardware as it requires the generation of partial products, which must be multiplied by a non-power of two factor. This issue was solved by Macsorley (1961), which proposed an algorithmic optimization by analyzing groups of 3 bits resulting in the multiples $\{0, \pm M, \pm 2M\}$ of the multiplicand. Hence, the number of generated partial products is equal to $\lceil n/2 \rceil$ if n is taken as the size of the multiplier.

Further algorithm optimizations have been proposed to optimize the hardware re-

alization of this multiplier. Bewick (1994) proposed a strategy for efficient sign propagation by adding three special bits to the first partial product, and two bits for the remaining ones, effectively removing the entire MSB replication required previously. Based on this strategy, Farooqui and Oklobdzija (1998) proposed a sign pre-computation technique, which resulted in a constant ($1010\ldots01011$) that should be inserted as a partial product, requiring only the first partial product to be sign-extended by one bit.

However, none of these techniques addressed the problem inherent the negative multiplicand multiples, which required that constant one was added on the LSB to correct the result as a consequence of the subtraction on two's complement. Själander and Larsson-Edefors (2008) introduced a solution to pre-compute the partial product LSB and insert and additional signal on the next bit position, reducing the size of the final recombination adder. Figure 4.6 illustrates an 8-bit Booth multiplier with all the aforementioned optimizations.

Figure 4.6: Modified-Booth multiplier with sign extension and LSBs pre-calculation

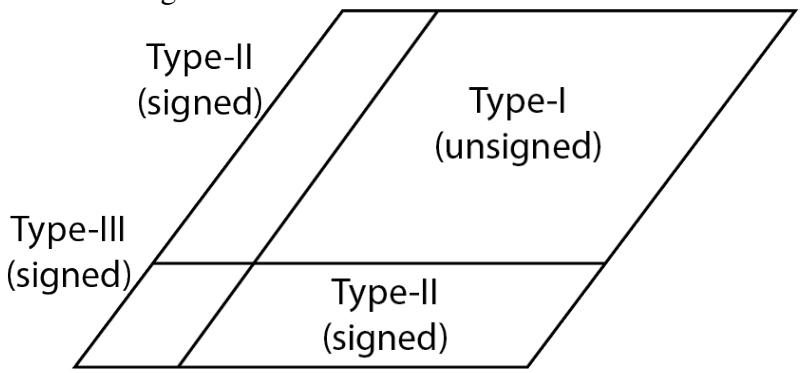


Source: (SJÄLANDER; LARSSON-EDEFORS, 2008)

4.2.1.2 Radix-2^m multiplier

A Booth-inspired partial product generation algorithm is the *Radix-2^m*, which was proposed by (COSTA; BAMPI; MONTEIRO, 2002) and it relies on splitting the n-bit input operands into groups of m bits that could be seen as individual multiplications. Hence, the n-bit multiplier is seen as a sum of several multiplications of m-bit digits where each multiplication can be of three types: (i) unsigned$\times$unsigned, (ii) signed$\times$signed, and (iii) signed$\times$signed. The multiplications that operate over the *W-m* least significant digits are unsigned while the remaining multiplications are signed, according to Figure 4.7. In this work, the scope is limited to the Radix-4 ($m = 2$) multipliers, although higher values of m follow the same approach.

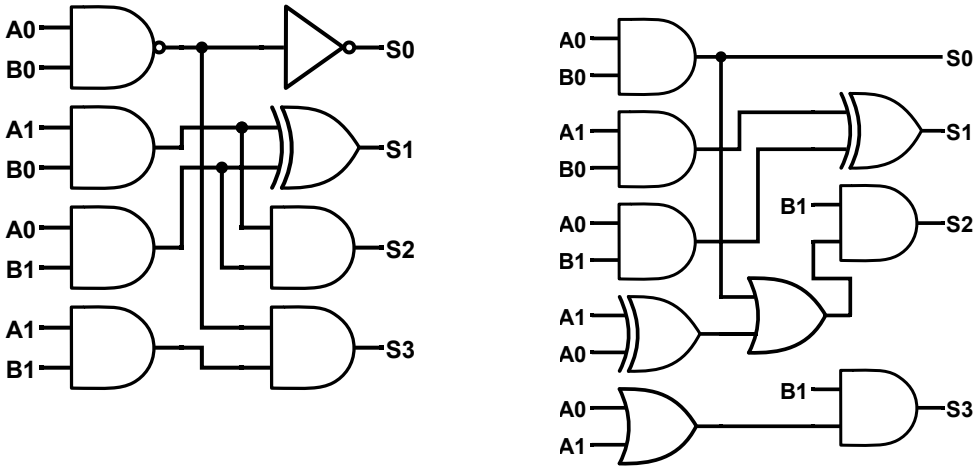
Figure 4.7: Radix-2^m encoder distribution



Source: The Author

There are three encoder types to implement the digit multiplication. The Type-I encoder (Figure 4.8a) implements the unsigned multiplication, and the Type-II (Figure 4.8b) encoder handles the signed×unsigned operation. Finally, the Type-III (Figure 4.9) encoder implements the multiplication of the MSB digits, which have the sign bit, requiring a more complex implementation. Note that only one Type-III encoder is required for the Radix-2^m regardless of the values of n and m. In all encoders, the $A0$ and $A1$ signals represent any two consecutive bits on the multiplicand, whereas $B0$ and $B1$ represent any two consecutive bits on the multiplier.

Figure 4.8: Type-I and II encoders on Radix-4 multiplier

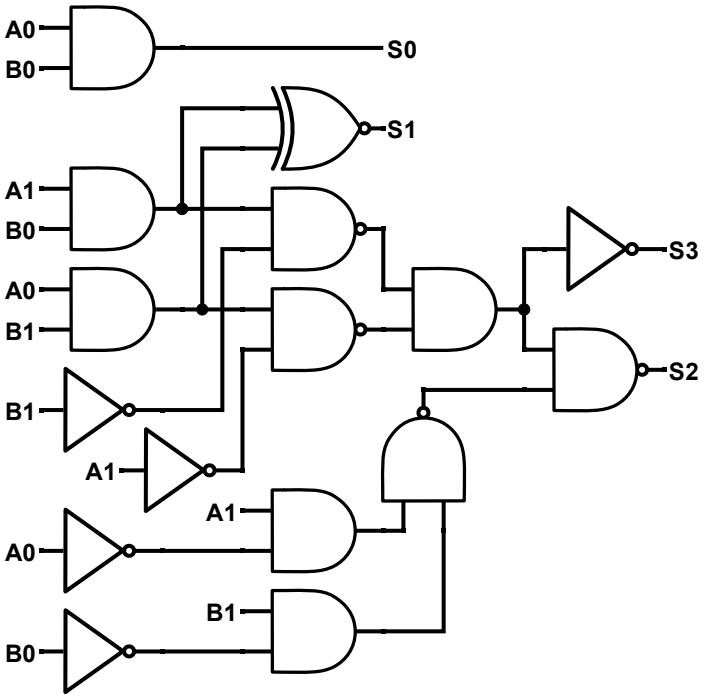


(a) Type-I encoder (b) Type-II encoder

Source: The Author

There are two types of partial products – *Operand I* and *Operand II* based on the three encoders available for the Radix-2^m multiplier. Originally, these operands were combined using a chain of RCAs instead of employing a carry-save approach, as illustrated in Figure 4.10. The two LSBs of each adder (except the last one) are concatenated with the two LSBs from Operand I to compose part of the final multiplication result.

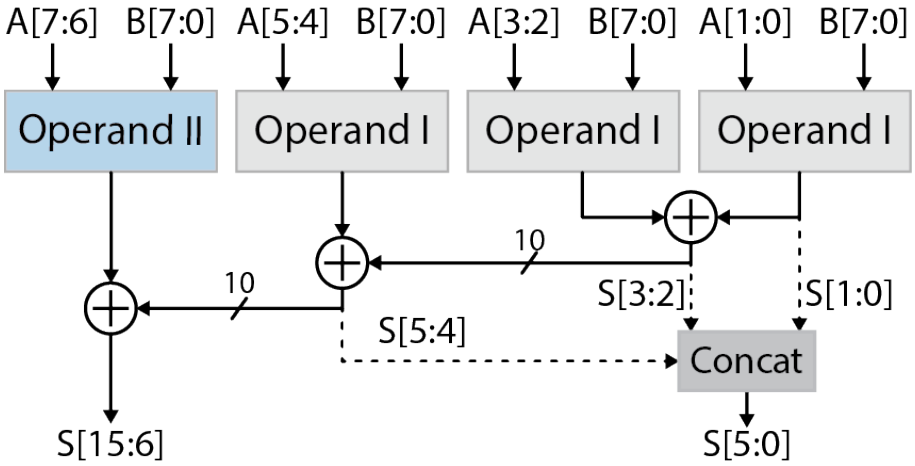
Figure 4.9: Type-III encoder on Radix-4 multiplier



Source: The Author

Each operand has a different construction approach. The Operand I, for instance, handle the unsigned×signed multiplication; hence it has $\frac{n}{m} - 1$ Type-I encoders (unsigned operation) and one Type-II encoder (signed operation), as illustrated in Figure 4.12. Since each encoder generates a 4-bit output for $m = 2$, the two output MSBs of encoder i is combined with the two output LSBs of encoder $i + 1$ using ripple-carry adders. Two exceptions arise in this operand: (i) the LSBs of the first Type-I encoded are directly copied to the output, and (ii) the MSBs are combined with the carry-out from the previous

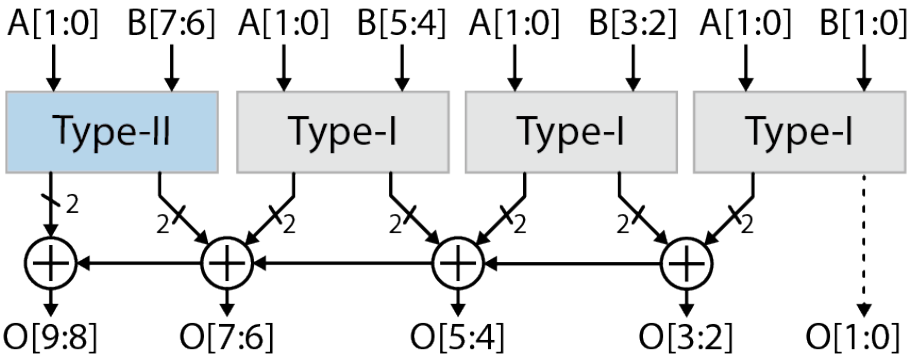
Figure 4.10: Partial product diagram for *W*=8 baseline Radix-4 multiplier



Source: The Author

adder. There are $\frac{n}{m} - 1$ Operand I partial products on a n-bit Radix-2^m multiplier.

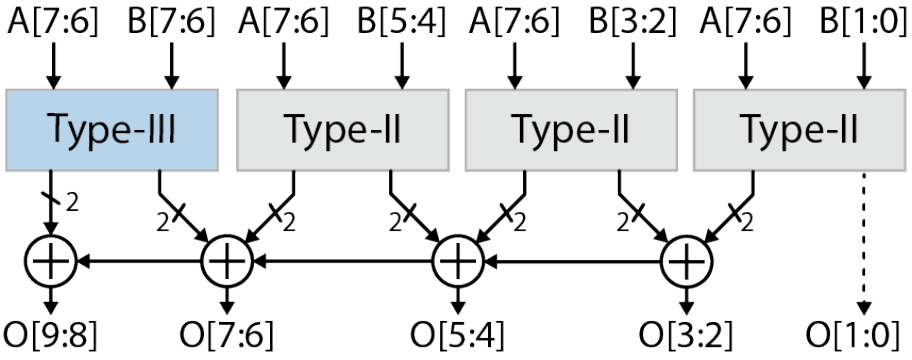
Figure 4.11: Internal structure of W=8 radix-4 operand I block



Source: The Author

The Operand II employs the same structure as Operand I, although it only handles signed operations. Hence, it features a single Type-III encoder along with $\frac{n}{m} - 1$ Type-II encoders, as illustrated in Figure 4.12. The encoders' outputs are combined exactly as in Operand I. Regardless of the multiplier size, only one Operand II partial product is generated.

Figure 4.12: Internal structure of W=8 radix-4 operand II block



Source: The Author

4.2.1.3 Baugh-Wooley Multiplier

Instead of recoding the partial products as in the Booth and Radix-2^m multipliers, the *Baugh-Wooley* algorithm aims for simpler hardware based on the unsigned array. This scheme also considers both multiplicand and multiplier to be informed in two's complement representations, although the partial products are positive, resulting in a simpler hardware (BAUGH; WOOLEY, 1973).

Reordering the partial products, Hatamian and Cash (1986) proposes a very regular structure to map this algorithm to hardware efficiently. Figure 4.13 shows the ar-

ray structure of a Baugh-Wooley multiplier, where x_n and y_n are the multiplicand and multiplier bits, respectively. Assuming A and X as the multiplicand and the multiplier, respectively, of arbitrary sizes, their multiplication is given by:

$$P = A \times X \tag{4.10}$$

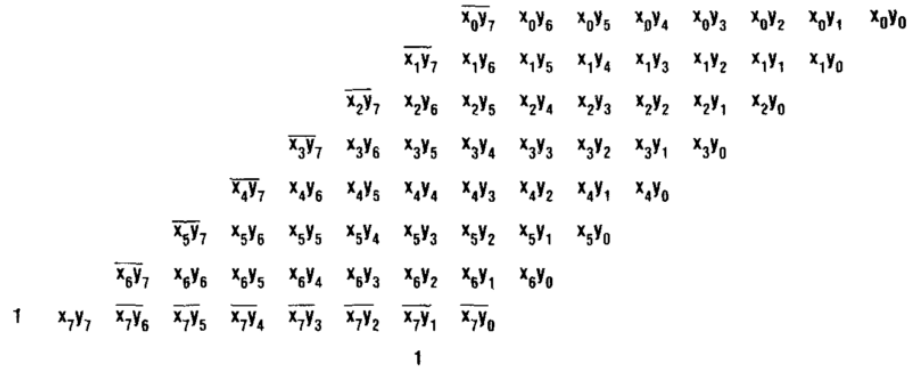
$$P = x_{n-1}a_{m-1}2^{n+m-2} + \sum_{i=0}^{n-2}\sum_{j=0}^{m-2} x_i a_j 2^{i+j} - x_{n-1}2^{n-1}\sum_{i=0}^{m-2} a_i 2^i - a_{m-1}2^{m-1}\sum_{i=0}^{n-2} x_i 2^i \tag{4.11}$$

The result obtained in (4.11) is obtained by applying the distributive multiplication property to (4.10). The two subtractions can be eliminated using the negation property in the two's complement representation, resulting in a homogeneous circuit that uses only adders. Hence, the multiplication can be described by:

$$2^{n-1}\left(-2^m + 2^{m-1} + \bar{x}_{n-1}2^{m-1} + x_{n-1} + \sum_{i=0}^{m-1} x_{n-1}\bar{y}_i 2^i\right) \tag{4.12}$$

For optimal hardware implementation, Hatamian and Cash (1986) proposed a partial product reordering, which leads to a very regular structure that disposes of the partial products according to Figure 4.13. This scheme generates n partial products that can be reduced using a carry-save tree. The lower partial product generation complexity leads to a considerably larger compression tree.

Figure 4.13: Partial products layout for the Baugh-Wooley algorithm



Source: (HATAMIAN; CASH, 1986)

4.2.2 Compression trees

Using binary adders to sum the partial products is not efficient in every aspect due to the carry propagation. Wallace (1964) proposed an adder tree based on a redundant representation (*sum* and *carry*) that operates without carry propagation, resulting in a much faster multiplier. The most popular carry-save trees are the Wallace and Dadda, both of them feature a latency that is logarithmically proportional to the number of partial products.

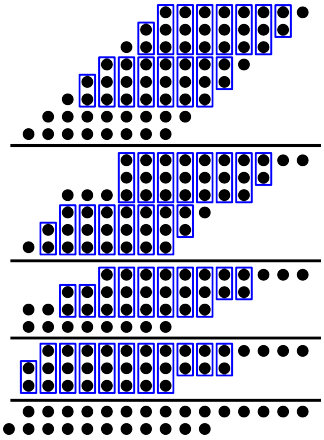
4.2.2.1 Wallace Tree

This approach aims to compress the partial products as much as possible in each level until there are only two rows of partial products to sum. The remaining signals are ready to be recombined by a carry propagating adders. Considering this scheme, the algorithm to generate such tree is as follows considering that only full and half adders are available as compression cells:

1. Take any group of three bits with the same weight and sum them using a full adder. If there are more bits of the same weight, group them with either a full or a half adder.
2. Propagate the outputs for the next stage whereas the sum bit will have the same weight i as the compressor inputs while the carry-out bit will have a higher weight $i + 1$.
3. If there is only a single bit left for a current weight, transfer it to the next level.
4. Repeat the steps above until there are no more than two bits left for any weight.

This algorithm is illustrated in Figure 4.14 for a generic 8×8 multiplier. Each blue rectangle represents a compressor cell that can be either a half or a full adder whereas each black dot represent either a signal from the partial product or an output of a previous compression cell. There is no connection between compressors on the same compression level, hence the critical path is given by the depth of the tree. In this case, the critical path is exactly four full adder cells.

Figure 4.14: 8×8 multiplier with a Wallace compression tree



Source: The Author

4.2.2.2 Dadda Tree

The Dadda tree avoids the greedy approach taken in the Wallace tree as it aims to reduce the number of operands in each stage using as few as possible compressor cells. This algorithm, proposed by Dadda (1965), is based a geometric progression that dictates the maximum number of summands on each stage of the tree. The available compressor cells define the tree compression ratio, i.e., the ratio that the geometric progression evolves. Considering that the full adder if the largest compression cell, the compression ratio is equal to$^3/_2$, and the sequence would follow the given pattern:

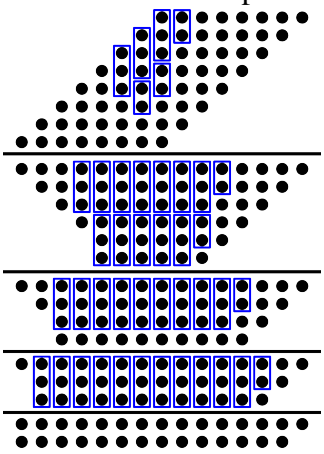
$$\overbrace{2}^{\text{Stage n}}, \underbrace{3}_{\text{Stage n-1}}, \overbrace{4}^{\text{Stage n-2}}, \underbrace{6}_{\text{Stage n-3}}, \overbrace{9}^{\text{Stage n-4}}, \underbrace{13}_{\text{Stage n-5}} \ldots \tag{4.13}$$

This algorithm is illustrated in Figure 4.15 assuming the same generic 8×8 multiplier as above. In the first level of compression, the longest column has eight bits, thus it has to be reduced to six bits to obey the progression shown above. For the subsequent levels, the same approach is used. Compared to the Wallace multiplier, this scheme uses less adders, resulting in a 26% smaller compression tree for an 8×8 multiplier. However, the final propagating adder size grows from 11 to 14 bits.

4.2.2.3 High-order Compressors for Multi-Operand Circuits

The adder compressors proposed by Wallace (1964) relied on full- and half-adder components, which could compress, at most, three inputs into two outputs. This carry-save idea was explored to use higher-order adder compressor cells to improve circuit

Figure 4.15: 8×8 Dadda multiplier reduction tree



Source: The Author

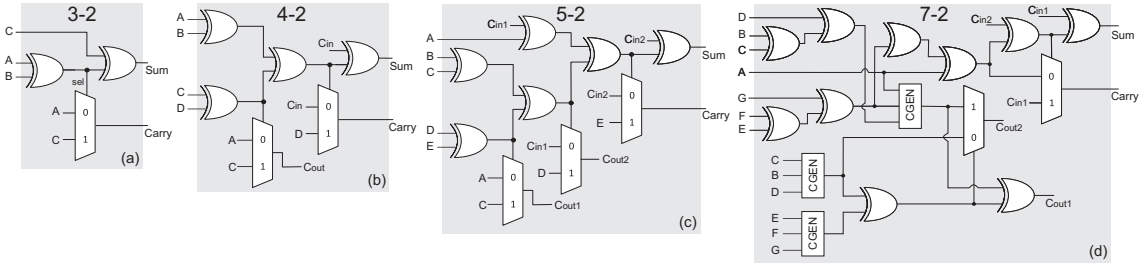
speed, area, and energy consumption. The first compressor was presented in Weinberger (1981), and it featured an arrangement to add four 1-bit inputs simultaneously. This structure was named 4-2 carry-save module, and it is composed of a combination of full adder cells in truncated connection so that a fast compression is possible. Oklobdzija and Krishnamurthy (2006) proposed for the first time a n-bit adder based on 4-2 adder compressors whose structure is shown in Figure 4.16b. These circuits often require a larger silicon area, but they benefit from a reduced dynamic power due to two characteristics:

- Well-designed compressors have limited critical path regardless of the bit-width of the operands. Consequently, the synthesis tool does not need to increase the cell strength to respect the timing constraints. This is particularly interesting for designs like floating-point multipliers and high-precision circuits.
- The internal structure of adder compressors and the limited signal propagation within the circuit are crucial factors to filter out spurious glitches that may occur due to the unbalanced timing path existent within a circuit.

More recent works explored high-order adder compressors like 5-2, 7-2, 8-2, 16-2 and so on, with emphasis in low-power design. Figure 4.16 summarizes the implementation of some elementary adder compressors. It is worth noting that the full-adder structure (4.16-a) is also known as the 3-2 compressor. In the 7-2 adder compressor structure proposed by Rouholamini et al. (2007) the critical path is given by six XOR gates. The 7-2 compressor has seven primary inputs, two carry inputs (*Cin*), two carry outputs (*Cout*), *Sum* and *Carry* output terms. Besides the XOR gates and MUX, the 7-2 compressor also uses a carry generator module (CGEN).

Basic adder compressors can be combined to perform a larger number of simul-

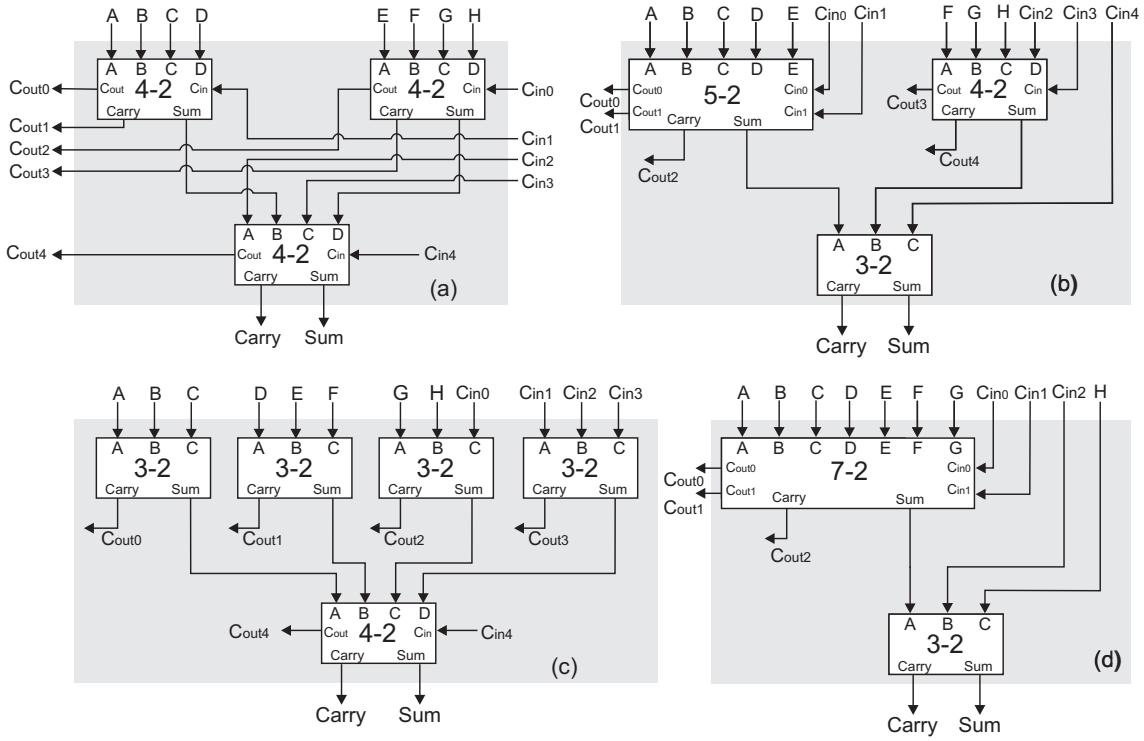
Figure 4.16: Adder compressors variants



Source: (SILVEIRA et al., 2017)

taneous additions. The 8-2 adder compressor, which allows the sum of up to 8 values simultaneously, can be implemented with different internal combinations of basic adder compressors. Recent works exploit four different internal structures for the hierarchical 8-2 adder compressor using combinations of basic 3-2, 4-2, 5-2 and 7-2 adder compressors. The four structures are shown in Figure 4.17.

Figure 4.17: Multiple 8-2 adder compressor structures



Source: (SILVEIRA et al., 2017)

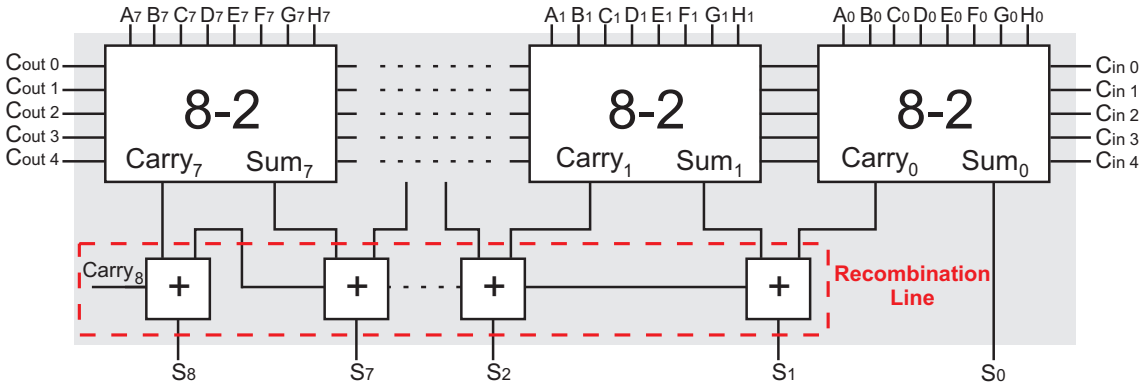
The result value is available in two primary outputs (*Sum* and *Carry*) and up to five carry outputs ($Cout_{0\text{-}4}$). The number of carry inputs and outputs depend on the basic adder compressors used to implement the 8-2 adder compressor. As an example, the 8-2

adder compressor composed of 7-2 and 3-2 adder compressors (Figure 4.17d) presents less carry inputs ($Cin_{0\text{-}2}$), and carry outputs ($Cout_{0\text{-}2}$). This occurs because the internal structure of 7-2 adder compressor is implemented with less input and output carries. The final sum result for the 8-2 adder compressor is given by:

$$S = Sum + 2(Cout_0 + Cout_1 + Cout_2 + Cout_3 + Cout_4 + Carry) \tag{4.14}$$

Figure 4.18 presents an addition of eight 8-bit values as an example. Note that carry-propagating adder circuits are required to recombine the partial sums of previous values (i.e. recombination line), since a *Carry* signal from the compressor *n* must be added with the *Sum* signal of the compressor *n+1* to generate the final sum (*S*) of bit *n+1*.

Figure 4.18: Sum of multiple 8-bit inputs

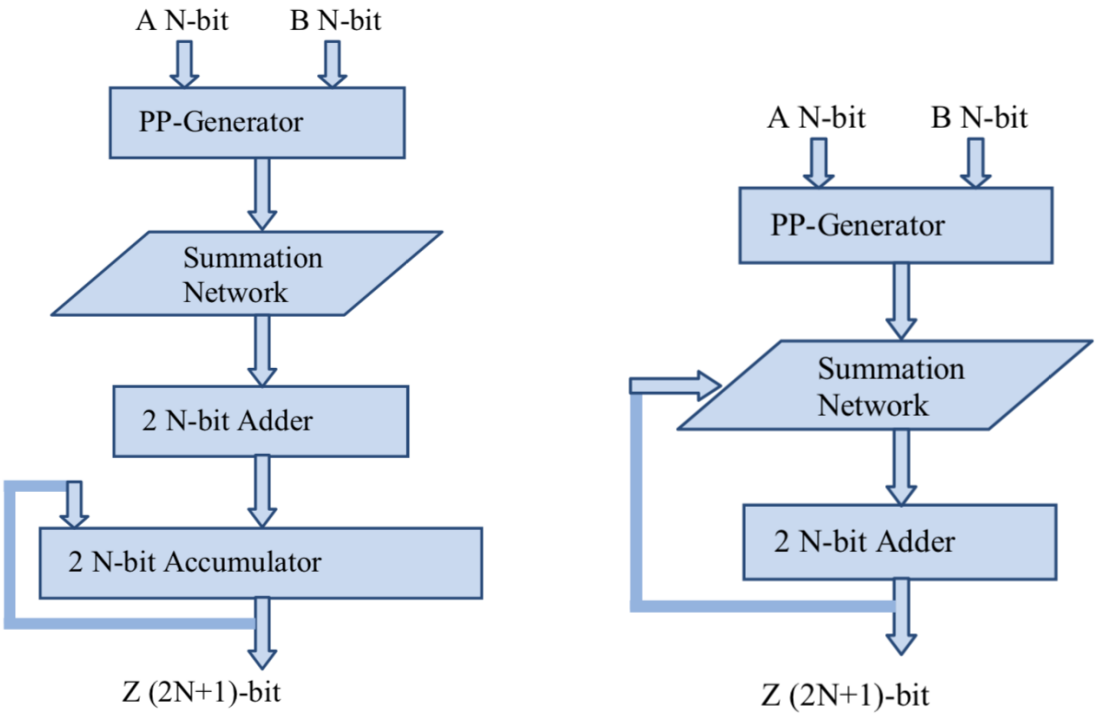


Source: (SILVEIRA et al., 2017)

4.3 Multiply-and-Accumulate (MAC) Units

Designing efficient MAC architectures have long presented several challenges that hardware designers must overcome to embed them into real-world applications. The basic MAC architecture includes a multiplier, an accumulator register, and an adder to sum these values. Some MAC architectures send the register output to the partial products compression tree to remove an additional carry-propagating adder (CPA) while others use a dedicated adder for this final sum (ABDELGAWAD; BAYOUMI, 2007). Figure 4.19a shows a basic MAC implementation where there are two CPAs, one to sum the partial products and others to sum the multiplication result with the accumulator. Conversely, Figure 4.19b embeds the accumulator directly into the compression tree, resulting in an architecture that requires only a single CPA adder.

Figure 4.19: Multiply-Accumulate Hardware Architecture



(a) Basic MAC Architecture (b) Fused Accumulator MAC Architecture

Source: (ABDELGAWAD, 2013)

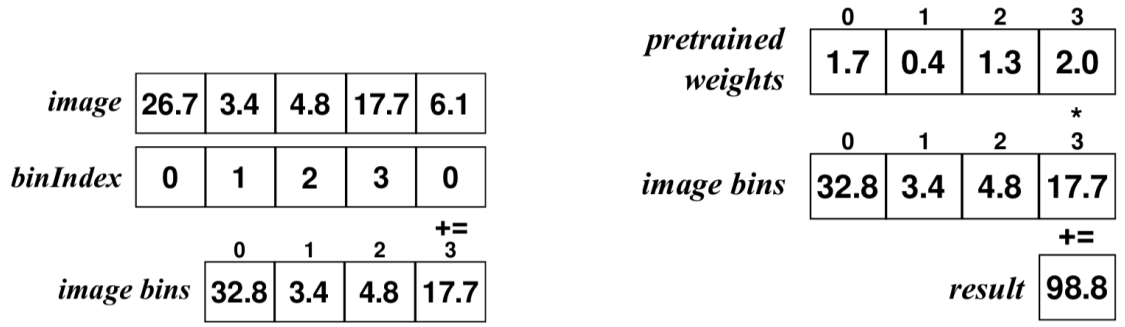
Early works like Stelling and Oklobdzija (1997) explored how to build MAC architectures with emphasis on speed efficiently. It is worth noting that the proposed design did not include the accumulator register as it received three inputs to operate $(A \times B) + C$.

Abdelgawad (2013) proposed a MAC unit with the accumulator value directly integrated into the compression tree to avoid employing two carry-propagating adders. Also, it featured a compression tree with adder compressors (see Section 4.2.2.3 for further details). Compared to the basic architecture, the fused MAC presents power savings up to 9% and delay improvement of 13%.

One of the earliest works to focus on the optimization of MAC units for CNN is found in (GARLAND; GREGG, 2017). The authors proposed an optimized MAC for convolutional neural networks based on weight sharing. This circuit counts each weight's frequency and accumulates the corresponding image value in a bin, replacing the multiplication logic by counters and selection logic. It contains b accumulators, one for each weight bin. As the input pixels stream into the MAC, the pixel value is added to the b_i container, which corresponds to the weight it should multiply (Figure 4.20a). Once all inputs have been processed, the b accumulators are respectively multiplied by the associated

weight values using a shared MAC unit (Figure 4.20b). The proposed MAC architecture consumes 70% less energy than a standard implementation when applied in this type of network.

Figure 4.20: Multiple computation phases on CNN-optimized MAC



(a) Phase 1: Weight frequency accumulation (b) Phase 2: Final MAC

Source: (GARLAND; GREGG, 2017)

4.4 Chapter Summmary

This Chapter presented a review on arithmetic circuits that are at the core of NN hardware accelerators. Optimizing these circuits is of utmost importance for the quest of energy-efficient accelerators for embedded devices. Recent works explored multioperand arithmetic circuits and different data representation schemes as promising approaches to build efficient NN execution kernels.

5 RTLGEN FRAMEWORK FOR ARITHMETIC KERNELS EXPLORATION

Most arithmetic structures are based on mathematical equations that describe how these modules are implemented, like carry-save adder trees. These equations often embed recursive operations whose translation to hardware circuits is not straightforward for the general case, i.e., for an arbitrary number of bits. Hardware description languages fail to provide full support to describe this genericity, relying on advanced language constructs that are often hard to understand and prone to errors. Further, despite the advancements in interpreter and synthesis engines in current electronic design automation tools, not all of them support these constructs, and the RTL interpretation may vary between vendors, leading to unsatisfactory design results.

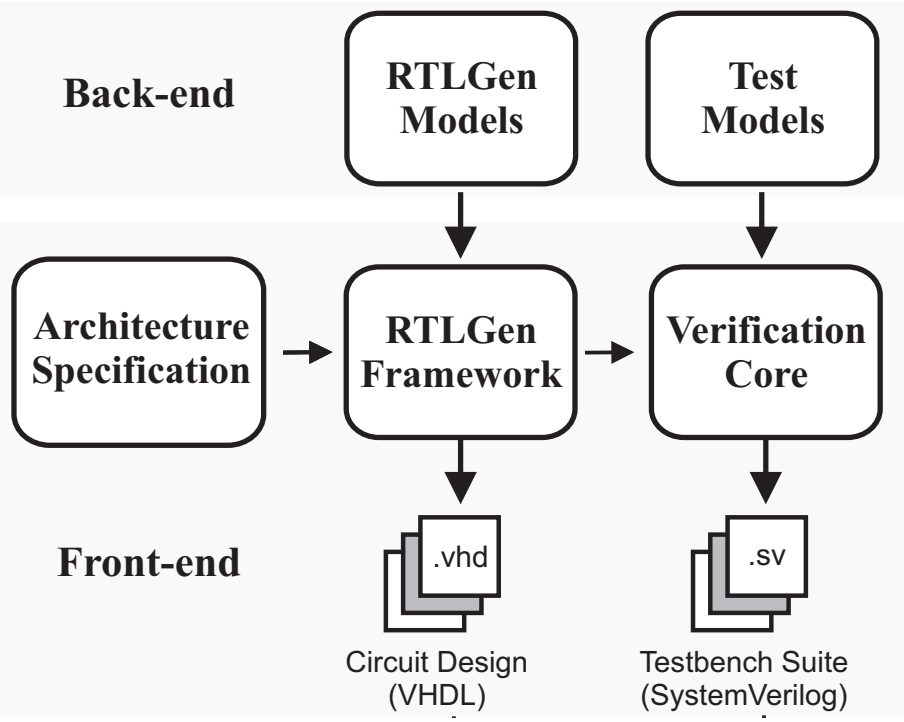
An efficient way to address this issue is to use hierarchical RTL designs based on simple language constructions. The authors in Rocha et al. (2017) proposed a framework named RTLGen that generates VHDL-based arithmetic circuits using a bottom-up strategy where the base modules are built with explicit logical equations without any advanced language features. The framework moves the design definition to another working space in a higher abstraction level where all complex constructions can be converted into multiple instantiations of simpler blocks, alleviating the burden on the interpreters of EDA tools. All circuit designs are described in Python, exploring all benefits of an object-oriented language, which is more readable and offers more resources to cope with elaborate designs. The framework is freely available for use by the community to generate custom arithmetic circuits on the following website: `<http://lmgrocha.pythonanywhere.com>`.

A key feature of RTLGen is that the generated circuits do not rely on any technology node, target platform, or synthesis tool. This interoperability is quintessential for digital designers as it enables granular control of the circuit – mandatory in critical applications like power-constrained environments and cryptographic circuits – and makes seamless the transition from a prototype in an FPGA platform to an ASIC-based synthesis flow.

Each generated circuit in RTLGen is self-contained, i.e., the generated file contains all the necessary components, instances, and specifications in a hierarchical VHDL design. This approach waives the necessity of custom libraries and constructs that determine the number of instances of a given module, signal width, among others, since all these parameters are computed before the RTL file generation. Further, each circuit has

an associated unit testbench for design verification and generation of support files for realistic power estimation in EDA tools. The design generation flow of RTLGen is illustrated in Figure 5.1 for a given set of architecture parameters.

Figure 5.1: Circuit generation flow of the RTLGen



Source: The Author

5.1 Framework architecture

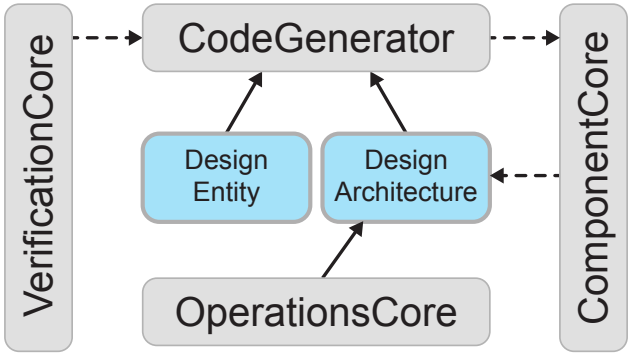
The RTLGen framework is divided into front- and back-end engines to generate reusable HDL-described circuits. This modularity has two key benefits: (a) it simplifies the framework extension to support new features and design constructs, and (b) it provides a simple and usable interface to generate and reuse the arithmetic circuits.

5.1.1 Back-end engine

The RTLGen back-end engine provides all the VHDL language constructs to describe the architectures in terms of logical equations, components, architectures, and entities. It also manages the integration of previously built components like partial product encoders, compression trees, carry propagating adders, and others into new designs, solv-

ing all dependencies to ensure that the RTL code of all modes will be properly embedded into the final design file. This engine is composed of four modules (*CodeGenerator*, *OperationsCore*, *ComponentCore*, *VerificationCore*) that are linked according to Figure 5.2. The *VerificationCore* will be explored in Section 5.1.2.

Figure 5.2: Back-end framework architecture



Source: The Author

The *OperationsCore* module maps all the VHDL constructs linked to combinational and sequential statements – logic equations, signal attributions, conditional statements, register declarations, and others – into Python classes. Internally, it manages input and output linking, multiple bit-width operands, and signal slicing before the RTL generation. There are three base classes that implement these functionalities: *Signal*, *Operation* and *Condition*. The *Signal* class is a placeholder for all circuit signals, while the *Operation* class implements all basic functionalities common to all combinational and sequential operations. Finally, the *Condition* class offers the foundation for all conditional statements, both inside and outside process statements. All these classes are further specialized to implement the particular functionalities of other VHDL constructs.

As the basic building blocks are created using the logical equations – a full adder, for instance – they can be reused in upper-level building blocks as simple instantiated objects. This procedure is managed by the *ComponentCore* module, which maps the previously conceived designs (full adders, carry propagating adders, compressors, etc.) as well as specific technology-related cells (isolation cells, mixed-signal components, etc.) into components. This mapping procedure translates each component into a Python class that can be further instantiated in any other portion of the circuit. All components are derived from the *Component* base class, which provides auxiliary functions required for component validation and instantiation, and each specialized class is a component that implements a specific architecture.

Once a component is instantiated in Python, the *ComponentCore* module manages

the procedures to instantiate the component on the RTL design. First, it ensures that the interface is properly linked, considering that the signals might have different bit-widths. It can also compute the number of modules to be instantiated when there is a bit-width mismatch between the linking signal and the interface. When the component has an associated RTL, this module ensures that the component definition is included in the final circuit design file.

All VHDL-based designs must have an entity, for interface definition, and an architecture that describes all the circuit functionality with its operations and components. The *CoreGenerator* module manages the link between the design architecture and the associated entity, and it handles the integration between the design and the testbench generation. The class *CodeGenerator* is the base class for each type of arithmetic circuit (adder, multiplier, MAC), and it internally generates the RTL code by sequentially checking all the operations and components belonging to the associated architecture.

Therefore, the *CoreGenerator* module has four primary responsibilities. First, it has to declare and instantiate all components used in the design, ensuring that all dependencies are met, and all components are appropriately instantiated and declared. Then, it has to declare all the internal signals used in operations and components, and it has to declare all combinational and sequential operations contained in the design model. Finally, it has to create and instantiate the test mechanism according to the design-specific parameters like test type, data source, and others.

5.1.2 Verification module

The RTLGen framework also offers an automatic unit testbench generation module (*VerificationCore*) which creates SystemVerilog test files based on the state-of-the-art Universal Verification Methodology (UVM) (ACCELERA ORGANIZATION, 2012). UVM is an open-source standard verification methodology widely supported by all major EDA tool vendors, and it enables creation interoperable and reusable verification IP and unit testbenches. Further, these unit tests can be configured to dump the switching activity into a specific file during the design simulation, which can be extremely helpful in power estimation planning procedures on the synthesis flow.

This module provides the functionalities to create the required components of a UVM-based verification unit. It takes as inputs a design previously built along with a set of pass/fail equations and the stimuli generation method. The module has two fac-

tory classes known as *BenchInterface* and *BenchStimulus*. The former manages the link between the VHDL-written circuit and the verification core written in SystemVerilog. It also controls whether or not the dump files will be created during the simulation. Conversely, the latter manages the implementation of the verification procedures according to the specified tests, and the generation/loading of the input stimulus for the design under test (DUT).

This approach simplifies the circuit verification as the conditions for the pass/fail tests can be declared in terms of standard SystemVerilog arithmetic operations (multipliers, adders, etc.) as well as in terms of direct comparison with golden values stored in external files generated by other applications or models. Support for external stimuli files is useful for non-standard operations like multiply-accumulate, approximate adders and multipliers, modular arithmetic, and others. Likewise, the external stimuli may contain values from real-case vector-like images, videos, and waves, which are needed to characterize the target design. Therefore, the verification module offers three test strategies:

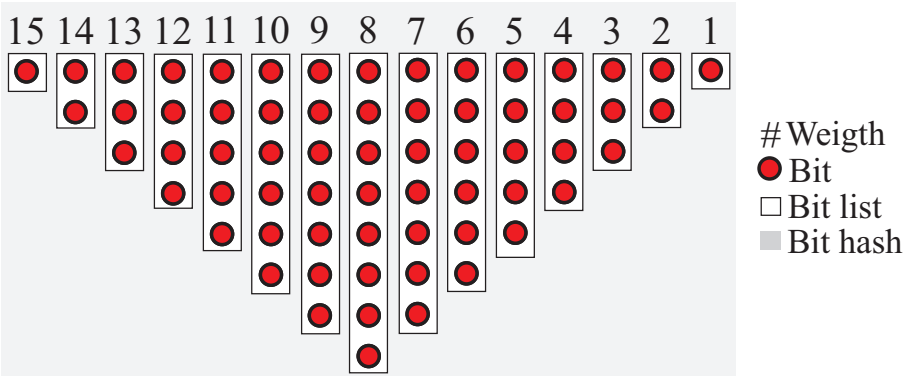
- **All-automatic strategy:** the simulation tool randomly generates the input stimuli. The user determines the number of input vectors, and the pass/fail equations use these values to compute the golden result to compare with the current circuit output.
- **Semi-automatic strategy:** the input stimuli are gathered from an external text file. These values are used to compute the golden result according to the pass/fail equations used for comparison with the circuit output. In this case, the simulation tool only needs to compute the golden result.
- **Manual strategy:** both the input stimuli and golden results are stored in external files. The pass/fail test only compares the current circuit output with the expected value. In this case, the simulation tool does not need to perform any computation.

When the testbench generation is called, the framework generates two files: (a) the interface, and (b) stimuli. The interface file instantiates the DUT, binding its inputs and output to the stimuli generator. It also manages the clock generation, which is parameterized, a fundamental aspect for post-synthesis simulations where timing restrictions apply. Conversely, the stimuli file creates the UVM environment and integrates the pass/fail tests according to the selected test strategy. This file is also responsible for providing either the external stimuli reading mechanism or routines to generate random input vectors.

5.1.3 Front-end engine

As the basic modules like full adders and multipliers encoders are described, the front-end engine provides a wrapper with the required functionalities to build more complex designs and the associated testbenches. The main feature of this engine is the bit-hash data structure – inspired on the work presented in (BRUNIE et al., 2013) – that holds a list of signals in specific weight indexes, handling all links among submodules used in the design. This bit-hash structure is based on a weight-aligned view where each column represents a bit weight, which increases from right to left. Although the bit-hash is not required for all arithmetic circuits, it provides a flexible and straightforward approach to manage all the interconnections efficiently, especially in multipliers and adder compressors.

Figure 5.3: Weight-aligned bit-hash for signal management



Source: The Author

Figure 5.3 illustrates how the bit-hash structure is populated based on an 8×8 unsigned array multiplier, which generates eight partial products of 8 bits. Each red circle represents a bit of a given partial product, and it is included in a list of signals on column i, representing the weight 2^i.

Algorithm 1 demonstrates the overall flow for generating a multiplier on the RTL-Gen framework using the functions provided by the front-end. In this case, the generation procedure requires a list of input operands (name and bit-width), the type of partial product generator, the compression tree type, and the desired carry propagating adder. In the end, the framework will output the RTL file containing all the statements.

The base design structures (entity and architecture) must be initialized (line 2) before the circuit can be defined. Then, line 2 creates the hardware that implements the partial product generation algorithm selected by the user (*PPGAlg*), and it populates the

Algorithm 1 Multiplier Generation Algorithm

Input: List of operands (A, B) and the desired algorithm for partial product generation, compression tree and carry propagating adder;

Output: Data structure containing all VHDL components and operations to realize the multiplier;

```
1:  Multiplier ← new Architecture
2:  PPG ← GeneratePP(PPGAlg, A, B)
3:  BitHash ← GetBitHash(PPG)
4:  for Operation, Component in PPG do
5:      AddOperation(Multiplier, Operation)
6:      AddComponent(Multiplier, Component)
7:  end for
8:  while max(LenCol(BitHash)) > 2 do
9:      Tree ← CompressTree(CompressAlg, BitHash)
10:     BitHash ← GetBitHash(Tree)
11:     for Operation, Component in Tree do
12:         AddOperation(Multiplier, Operation)
13:         AddComponent(Multiplier, Component)
14:     end for
15: end while
16: Adder ← new CPAdder(AdderType,BitHash)
17: AddComponent(Multiplier, Adder)
```

The Multiplier structure may now be printed to a regular text file that will contain the VHDL description.

bit-hash accordingly. Once the partial products are generated, the bit-hash is recuperated (line 3), and the operations and components that belong to the partial product generator are added to the top-level architecture (lines 5 and 6).

Since the bit-hash has columns with more than two signals, it must be reduced before the carry propagating adder can be instantiated. Hence, lines 8 to 15 iterates over the data structure until all columns have, at most, two elements. The compression (line 9) applies a carry-save-based tree compression algorithm, like Wallace or Dadda, according to the specified parameter (*CompressAlg*), and it returns the newly added operations and components for that compression level. The bit-hash is updated (line 10) with the new version containing the remaining signals. Once the compression algorithm is finished, the carry propagating adder is created and instantiated in lines 15 and 16, respectively, according to the *AdderType* parameter. After all these steps, the final multiplier design is ready to be exported to a text file.

In each tree compression iteration, the *CompressTree* function executes several internal procedures. First, it selects the signals that will be removed from the bit-hash and connected to the compressor cell. The signal selection depends on the selected tree

compression algorithm. The signals not selected in this step are ignored and kept on the bit-hash, preventing carry propagation within the same stage. This approach guarantees that the delay for each compression level is exactly one compressor cells. Once these steps are done, the compressor outputs are inserted into bit-hash to be properly connected on the next iteration or CPA connection.

5.2 Framework Evaluation

Since RTLGen embeds a UVM-based verification core, all generated circuits are guaranteed to be functionally correct. Therefore, the framework can only be evaluated in terms of improvements in the project time-to-market and team resources optimization. The framework benefits are evaluated considering a small set of multipliers, limiting to three partial product generation algorithms (Modified-Booth, a modified version of the Radix-2^m(COSTA; BAMPI; MONTEIRO, 2002) and Baugh-Wooley), two compression algorithms (Wallace and Dadda), and three recombination line adders (synthesis tool-inferred adder, carry select and Kogge-Stone). The tool-inferred multiplier based on the * VHDL operator is also included in the comparison.

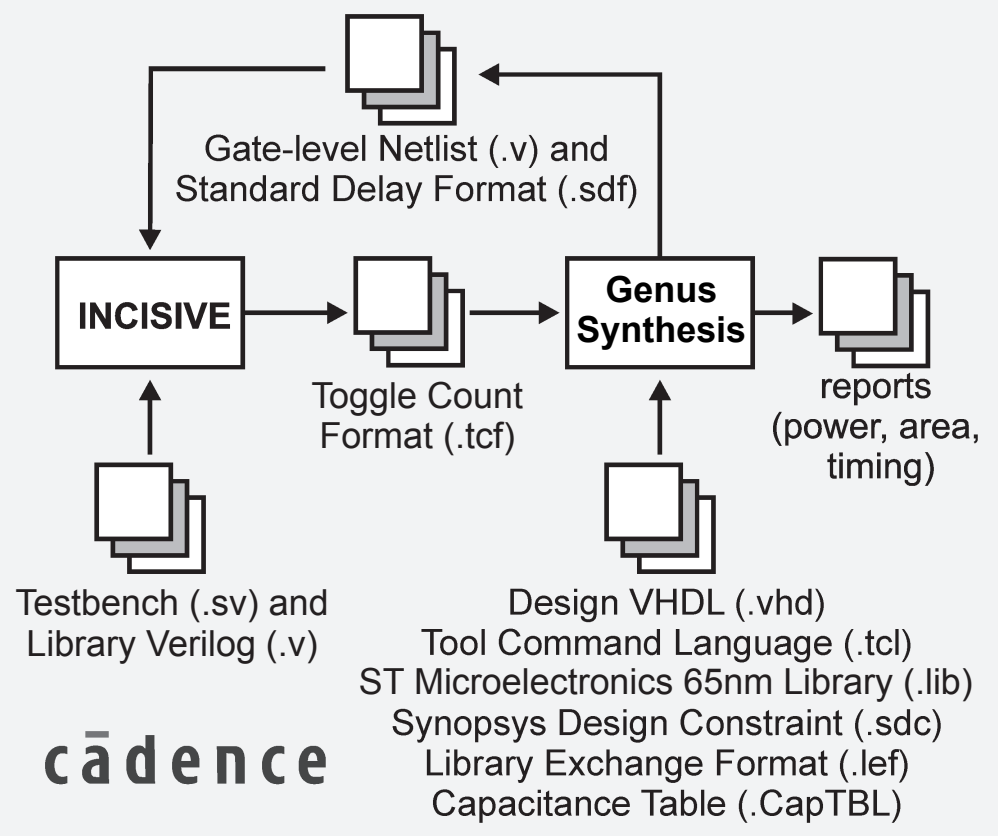
Some arithmetic architectures are more suited for smaller data sizes due to their complexity, while others may have an excellent performance in larger datapaths due to their scalability. The multipliers mentioned above were generated for inputs with 8, 16, 32, and 64 bits to cover all possibilities. Performing this circuit generation task by hand is impractical due to the number of designs (72, in total) even for an experienced digital designer. This task is solved in RTLGen using iterative loops, and it is executed in a short period with few resources. This design exploration has main usages: (a) it enables the analysis of multiple architectures under the same constraints, and (b) it helps on the characterization of standard cell libraries.

5.2.1 Power Extraction Methodology

Despite the advances in EDA tools, their power estimation methodology on synthesized netlists is far too pessimistic as it assumes a probabilistic switching activity on all nodes, and it does not model the effects of signal propagation on power dissipation. This issue is solved by exploring the interoperability of the testbench generated by the

RTLGen framework using the all-automatic testbench generation strategy. The synthesized netlist is simulated with the input vectors generated by the testbench, stimulating all circuit nodes. This simulation also considers the interconnection delay among gates through the Standard Delay Format (SDF) file generated by the synthesis tool. The circuit activity is dumped by the simulation tool into stimuli files like Value Change Dump (VCD), Toggle Count Format (TCF), or Switching Activity Interchange Format (SAIF). These files are fed to the synthesis tool along with the previously synthesized netlist for the accurate power estimation.

Figure 5.4: Synthesis and simulation flow



Source: (PAIM et al., 2019a)

This methodology is similar to the one presented in Paim et al. (2019a). Initially, all RTL designs are synthesized using the Cadence Genus Synthesis (CADENCE, 2018) tool with the Physically-Aware Layout Estimation (PLE) mode enabled. This mode performs primitive floorplanning to estimate the wire length to connect all the digital cells, and it is quintessential to take into account the effects of these wires in terms of the critical path, area, and power consumption. The netlist simulation is executed by the Cadence Incisive Simulation Tool. The overall flow is illustrated in Figure 5.4.

5.2.2 Multipliers Architectural Exploration with RTLGen: Synthesis Results

In this subsection the results obtained by synthesizing dozens of multipliers (of 8×8 to 64×64) is being demonstrated. All multipliers were generated by either RTLGen or by the commercial tool own inferred multiplier for a given frequency – this latter for the sake of comparison. All circuits were synthesized with an ST Microelectronics 65 nm standard cell library operating at 1.0V supply voltage. A size-based sweep methodology is adopted to find the maximum operating frequency for each input bit-width. Since the main goal is to explore the RTLGen framework's versatility, the synthesis tool is set to use the standard synthesis run parameters.

Table 5.1 shows a comparison of all multiplier versions for multiple input bit-widths whose results were obtained at slack zero (maximum operating frequency). The table considers the maximum operating frequency (F_{max}), circuit cell area (C. Area), dynamic power dissipation (D. Power), and total power dissipation (T. Power) for all the combinations of partial products processing, compression trees, and carry propagating adders. The synthesis tool multipliers are used as baseline models for comparison.

Although the partial product algorithms substantially impact the quality of results (QoR) of the synthesized circuit, it is overwhelmed by the compression tree selection in all aspects (speed, area, and power). For instance, all multipliers using Dadda trees presented better results than their Wallace counterparts in almost all cases. On average, Dadda-based circuits offer 3.3% higher F_{max} and 9.3% smaller cell area even at higher speeds.

Despite the proposal of newer partial product encoders, the Modified Booth multiplier outperformed all the considered algorithms in every aspect. This outstanding performance, compared with its peers, is due to the high level of optimization integrated into this architecture, from LSB pre-computation to sign propagation optimization. Despite the original claims in Costa, Bampi and Monteiro (2002), the Radix-4 showed the worst performance because it lacks optimization.

Theoretically, all Kogge-Stone-based multipliers should present the highest maximum operating frequencies. However, this assumption does not hold as the wire effects are considered by the synthesis tool, so the wire congestion can be a bottleneck that penalizes the circuit, as seen in the specific case of Table 5.1. For instance, the 16-bit Booth multiplier using Dadda tree and carry select adder is slightly faster and is 5% smaller than the version using the Kogge-Stone adder.

Table 5.1: Circuit Speed, Area and Power Dissipation Comparison @ Maximum Frequency and Worst-case PVT conditions

			Wallace Tree				**Dadda Tree**			
Size	**Mult.**	**Adder**	F_{max} (MHz)	C. Area (μm^2)	D. Power (μW)	T. Power (μW)	F_{max} (MHz)	C. Area (μm^2)	D. Power (μW)	T. Power. (μW)
8×8	Array-Uns	**Tool**	452.7	1890.2	749.0	751.5	499.7	1962.0	917.7	920.2
		C-Select	451.3	2096.1	823.6	826.5	485.8	2212.6	1012.8	1015.8
		K-Stone	451.8	2226.6	870.4	873.5	412.1	1855.4	721.8	724.2
	M-Booth	**Tool**	462.5	2450.8	1158.1	1161.7	501.0	2134.6	1273.7	1276.7
		C-Select	467.3	2647.3	1243.6	1247.6	497.2	2260.4	1171.5	1174.7
		K-Stone	475.3	2840.8	1252.5	1256.7	495.7	2435.2	1263.7	1267.3
	Radix-4	**Tool**	471.7	3571.4	1372.8	1378.0	475.0	2993.1	1195.9	1200.0
		C-Select	443.4	3138.2	1177.5	1181.8	452.0	2752.9	1121.8	1125.6
		K-Stone	454.5	2918.2	1129.9	1134.0	476.2	3288.0	1243.8	1248.6
	B-Wooley	**Tool**	451.8	1763.8	803.7	806.0	489.6	2068.0	979.0	981.7
		C-Select	452.2	2135.6	957.6	960.6	469.6	2126.3	1001.4	1004.3
		K-Stone	454.5	2032.2	945.7	948.6	463.1	1966.6	913.9	916.5
	S. Tool	N/A	401.7	1612.0	709.5	711.6				
16×16	Array-Uns	**Tool**	380.5	8904.9	2531.5	2542.8	383.3	7100.1	2274.1	2282.5
		C-Select	352.4	9483.8	2577.6	2589.8	356.4	7664.8	2221.5	2230.8
		K-Stone	357.1	8637.2	2356.5	2367.4	375.5	8087.0	2547.4	2557.5
	M-Booth	**Tool**	396.9	8091.7	2898.2	2909.6	403.0	8522.3	3282.8	3294.6
		C-Select	392.8	9111.4	3265.4	3278.4	398.1	8263.8	3202.1	3213.9
		K-Stone	390.2	9571.1	3411.4	3425.2	395.1	8700.1	3207.9	3220.1
	Radix-4	**Tool**	365.0	12600.1	3395.8	3412.1	365.4	10657.4	3161.4	3174.7
		C-Select	357.1	13119.6	3432.7	3450.0	366.5	12060.4	3417.9	3433.7
		K-Stone	352.4	12679.2	3381.6	3398.1	367.8	12212.7	3478.1	3494.0
	B-Wooley	**Tool**	381.3	9107.3	2633.1	2644.7	374.4	6505.7	2151.0	2158.5
		C-Select	366.0	9594.4	2789.6	2801.9	372.1	7911.3	2532.3	2542.2
		K-Stone	369.3	9814.0	2794.1	2806.9	370.4	7437.6	2369.6	2378.5
	S. Tool	N/A	395.2	7142.2	2684.2	2693.9				
32×32	Array-Uns	**Tool**	318.1	35870.1	7845.9	7890.1	322.8	28901.1	7288.1	7321.3
		C-Select	290.9	35885.2	7230.9	7274.4	297.9	31844.8	7165.4	7203.7
		K-Stone	292.6	36265.3	7264.9	7308.5	316.2	34193.1	8179.1	8220.5
	M-Booth	**Tool**	321.4	29599.4	8119.4	8159.4	331.5	26371.3	7755.7	7789.5
		C-Select	308.8	31203.6	8105.6	8149.0	322.6	28861.6	8008.7	8048.3
		K-Stone	311.3	31944.6	8241.3	8284.6	324.5	29484.5	8636.5	8675.6
	Radix-4	**Tool**	300.2	44667.0	9564.8	9618.6	293.8	36141.6	8164.8	8204.8
		C-Select	275.5	43820.4	8702.3	8754.7	289.6	40287.0	8973.6	9021.1
		K-Stone	285.3	43739.8	8921.2	8973.0	293.2	40943.2	9123.8	9172.1
	B-Wooley	**Tool**	311.1	35196.2	7586.0	7629.1	325.7	30972.8	7771.0	7807.7
		C-Select	283.1	34287.8	6810.8	6852.2	298.1	32492.7	7336.2	7375.3
		K-Stone	282.6	32685.1	6530.9	6570.1	318.2	34278.9	8238.2	8280.2
	S. Tool	N/A	309.3	22642.9	6596.9	6614.5				
64×64	Array-Uns	**Tool**	267.1	133899.5	23567.4	23725.3	277.2	107101.3	20835.5	20954.0
		C-Select	238.1	128600.2	19900.2	20049.4	250.0	110924.8	19177.5	19300.1
		K-Stone	259.7	139051.6	23674.0	23838.7	272.3	117946.4	23352.0	23486.3
	M-Booth	**Tool**	268.9	102998.0	21533.9	21663.3	281.3	95656.1	22675.2	22791.1
		C-Select	243.7	104598.0	19983.4	20115.8	255.7	98441.2	20418.2	20543.2
		K-Stone	260.6	109949.3	22333.9	22472.3	268.9	99602.4	22138.7	22262.3
	Radix-4	**Tool**	262.1	163242.7	29093.1	29284.9	258.1	145539.7	26902.6	27067.9
		C-Select	246.1	172206.8	28994.0	29202.1	242.5	159011.8	26542.3	26729.1
		K-Stone	258.2	177336.1	32195.2	32406.6	257.4	168142.0	31122.2	31321.4
	B-Wooley	**Tool**	267.3	128695.8	22744.6	22894.6	281.6	114199.8	23171.1	23301.5
		C-Select	243.5	130369.7	20772.5	20925.5	250.0	113492.1	19598.3	19726.8
		K-Stone	254.2	129878.8	21729.9	21881.4	270.0	115556.5	22140.9	22270.7
	S. Tool	N/A	267.5	85302.4	18903.8	19004.9				

Source: The Author

Table 5.2: Synthesis QoR Comparison at same frequency of operation.

			Wallace Tree			Dadda Tree		
Size	**Mult.**	**Adder**	C. Area (μm^2)	D. Power (μW)	T. Power (μW)	C. Area (μm^2)	D. Power (μW)	T. Power. (μW)
8×8[1]	Array-Uns	**Tool**	1566.8	595.4	597.3	1635.4	601.0	603.0
		C-Select	2057.1	761.4	764.2	2225.6	774.9	778.0
		K-Stone	1880.8	674.7	676.9	2350.9	727.3	730.6
	M-Booth	**Tool**	1603.2	675.8	677.7	2261.0	888.9	892.0
		C-Select	2195.4	941.7	944.7	2478.3	994.9	998.3
		K-Stone	2019.7	812.2	814.7	2516.3	954.3	957.9
	Radix-4	**Tool**	2136.7	773.6	776.2	2724.3	889.0	892.6
		C-Select	2619.8	1037.0	1040.4	3125.2	1134.4	1138.8
		K-Stone	3086.7	1062.7	1066.7	2944.2	1017.9	1021.9
	B-Wooley	**Tool**	1694.2	669.9	671.9	1711.8	720.2	722.5
		C-Select	2160.6	924.3	927.3	2183.0	826.9	829.8
		K-Stone	2050.4	788.5	791.2	2433.1	815.7	819.2
	S. Tool	N/A	1663.5	814.3	816.5			
16×16[2]	Array-Uns	**Tool**	4346.7	840.6	844.6	4599.4	786.3	790.2
		C-Select	4765.3	933.7	937.8	5181.8	844.1	848.8
		K-Stone	3933.3	872.7	875.0	4397.6	783.7	786.6
	M-Booth	**Tool**	4318.1	1072.1	1076.1	4694.6	1123.1	1127.6
		C-Select	4705.0	1176.5	1180.7	4887.5	1102.7	1107.2
		K-Stone	4471.0	1121.0	1124.3	4784.5	1223.0	1226.9
	Radix-4	**Tool**	5542.2	1001.4	1006.0	6408.0	1031.8	1037.9
		C-Select	6128.7	1055.9	1061.5	6526.5	1081.2	1087.1
		K-Stone	5386.7	1034.3	1037.7	5303.5	1029.7	1032.9
	B-Wooley	**Tool**	4339.9	814.4	818.0	4637.9	795.4	799.3
		C-Select	4585.9	870.7	874.5	5157.9	861.8	866.5
		K-Stone	4001.9	909.9	912.3	4422.6	815.6	818.5
	S. Tool	N/A	4004.0	934.6	938.4			
32×32[3]	Array-Uns	**Tool**	13600.1	1535.8	1543.7	15838.2	1604.2	1615.2
		C-Select	16759.6	2120.6	2133.4	18066.4	1825.7	1840.1
		K-Stone	14791.4	2344.6	2352.6	15358.2	1972.4	1980.6
	M-Booth	**Tool**	16138.2	2045.9	2059.5	16860.0	2275.0	2289.4
		C-Select	17155.3	2736.0	2751.0	18426.2	2485.7	2502.6
		K-Stone	16293.2	2952.7	2964.8	17084.1	2640.7	2653.3
	Radix-4	**Tool**	18439.2	1867.0	1877.3	19914.4	1912.3	1925.0
		C-Select	21792.2	2176.3	2194.0	22954.4	2238.1	2256.7
		K-Stone	19338.8	2480.2	2490.2	19615.4	2198.2	2208.2
	B-Wooley	**Tool**	13229.8	1565.5	1572.7	15778.4	1596.1	1607.0
		C-Select	16574.5	1921.3	1933.9	18015.4	1840.8	1855.2
		K-Stone	14455.5	2328.1	2335.6	15116.9	1978.5	1986.5
	S. Tool	N/A	13136.2	3088.0	3098.9			
64×64[1]	Array-Uns	**Tool**	51385.9	4012.2	4038.2	54114.3	4132.3	4160.6
		C-Select	64745.7	5061.8	5111.1	66808.6	4800.7	4851.4
		K-Stone	54457.0	5780.3	5807.1	56538.5	5226.9	5254.9
	M-Booth	**Tool**	58032.0	6450.0	6495.0	64734.3	5765.4	5820.4
		C-Select	64599.1	6934.9	6991.5	69560.9	6523.5	6587.4
		K-Stone	63822.7	8074.5	8123.3	62884.6	7570.1	7616.8
	Radix-4	**Tool**	70862.5	4695.7	4734.2	72205.6	4660.5	4701.8
		C-Select	86803.6	6144.1	6210.1	85311.2	5740.2	5803.9
		K-Stone	73161.9	6452.3	6488.9	73372.5	5977.2	6013.8
	B-Wooley	**Tool**	51506.5	4158.7	4185.0	54276.0	4054.3	4082.6
		C-Select	63635.5	4849.8	4896.9	66435.2	4820.7	4870.4
		K-Stone	54552.2	5767.3	5794.2	56642.0	5170.1	5198.2
	S. Tool	N/A	56331.6	16262.2	16317.3			

[1] Target frequency 400 MHz [2] Target frequency 200 MHz [3] Target frequency 133 MHz
[4] Target frequency 100 MHz

Source: The Author

It is worth mentioning that a direct comparison of power and area in Table 5.1 is not fair. The multipliers at higher frequencies not only will require larger cells to ensure the transition time, but also the dynamic power is directly proportional to the clock frequency. Therefore, each group of multipliers with the same input bit-width was synthesized, aiming at a specific operational frequency. The target synthesis frequencies for multipliers with input sizes 8, 16, 32, and 64 bits are 400 MHz, 200 MHz, 133 MHz, and 100 MHz, respectively. The selected frequencies are lower than the maximum attainable frequencies for two reasons: first, it reduces the overall circuit fan-out, ensuring that all circuits respect the timing constraints. These results are shown in Table 5.2 and indicate the cell area (C. Area), dynamic (D. Power), and total power (T. Power) dissipation for each multiplier. In all cases, the tool-inferred multipliers are outperformed by all RTLGen designs in all aspects.

Results in in Table 5.2 shows the same tendencies as the ones presented in Table 5.1. For instance, Dadda-based multipliers outperform Wallace-based multipliers in nearly all situations. The RTLGen benefits are evident, especially in higher bit-width multipliers (32 and 64 bits) as the synthesis tools endeavor does not lead to good results in that particular frequency range. In the case of the 32-bit tool-inferred multiplier, the Baugh-Wooley circuit with Wallace tree and a tool-inferred adder dissipates 48% less power with an area penalty about 20%. For the 64-bit version, the power dissipation savings are even more substantial as the tool-inferred multiplier dissipates nearly four times more than the best circuit generated by RTLGen. In this case, the best result is achieved by a Baugh-Wooley partial product generator associated with a Dadda compression tree and a tool-inferred adder.

The flexibility, extensibility, and interoperability offered by RTLGen along with the promising results, indicate the framework efficiency when determining the most suitable architecture for a constraint set. Further, most EDA tools do not focus on offering an optimized mapping method for multiplier circuits, which may negatively impact the QoR in larger projects.

5.3 Case study: optimized Radix-2^m multiplier

The flexibility of RTLGen allows the optimization of state-of-the-art arithmetic operators. As a case study, the Radix-2^m multiplier is a suitable choice for optimization as recent works like Pieper, Costa and Monteiro (2013) and Martins, Fonseca and Costa

(2015) do not propose an efficient sign propagation, a key element on multipliers' performance. The proposed architectures rely on replicating the most significant bit to match the alignment of partial products as well as intermediary ripple-carry adders on the partial product generation, which severely increases the circuit's critical path.

5.3.1 Efficient Signal Extension Method for Radix-2^m Parallel Multiplier

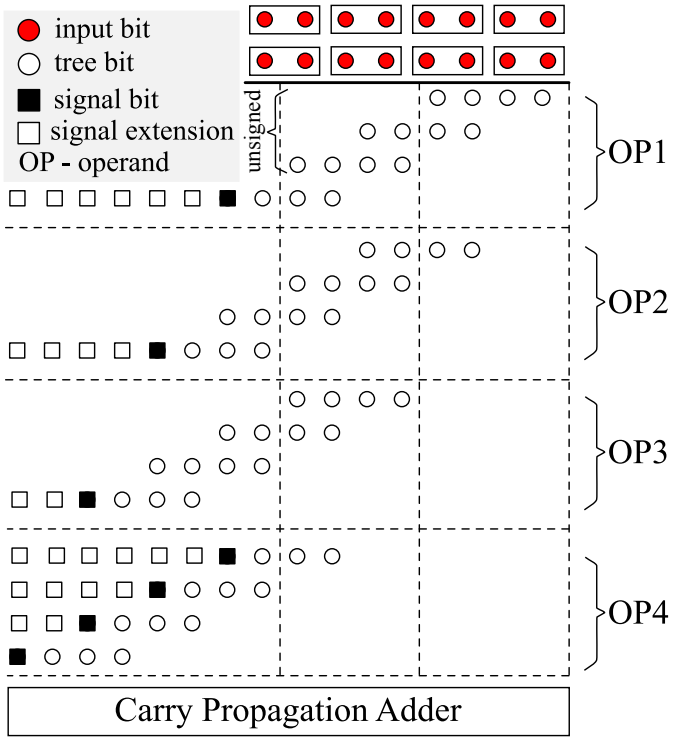
All Radix-2^m multiplier architectures proposed so far have limited power efficiency and maximum operating frequency due to the adoption of internal RCAs and naïve sign-extension. Hence, this work proposes a suitable optimization strategy to improve the circuit QoR based on two approaches: (i) replacement of all intermediary RCAs for a carry-save approach, allowing the adoption of a compression tree, and (ii) pre-computation of the sign bits to avoid the MSB replication. This optimization targets Radix-2^m multipliers for $m = 2$, although this strategy can be scaled for other values of m.

Removing the intermediary RCA requires aligning all encoder outputs according to their respective bit weight, as illustrated in Figure 5.5 for an 8-bit multiplier. Note that only the bits represented by a black square need to be extended as they convey the sign information for that particular operand. This approach can be directly mapped to the bit-hash structure on RTLGen to build a compression tree based on Wallace or Dadda algorithms. Although the RCA substitution for a compression tree leads to a smaller area and critical path due to fewer components, it still requires replicating the MSB of the signed encoders.

$$x = -x_{n-1} + \sum_{i=0}^{n-2} x_i 2^{-i} \tag{5.1}$$

$$(-x_{n-1}) + 1 - 1 = (1 - x_{n-1}) - 1 = \bar{x_{n-1}} - 1 \tag{5.2}$$

Figure 5.5: Weight-aligned partial products without RCAs



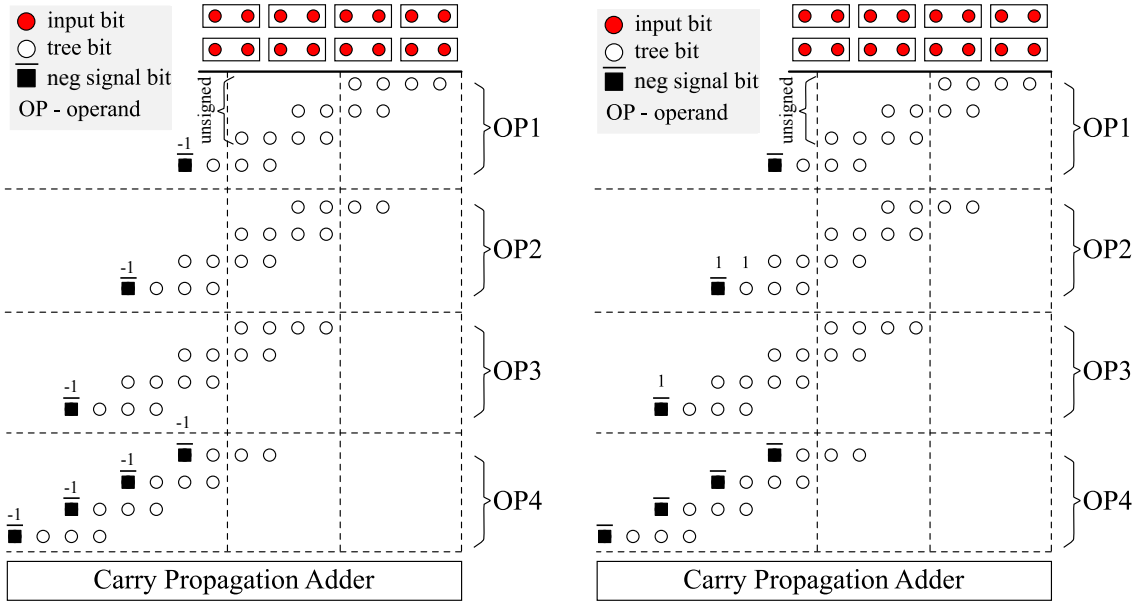
Source: The Author

The sign bit extension can be optimized by exploring the properties of the two's complement representation (5.1) based on the approach adopted by Farooqui and Oklobdzija (1998). In this case, all signed partial products are assumed to be negative, according to (5.2), so their representation has an infinite sequence of ones from right to left starting on the negated version of the MSB. If the partial product is negative, the sum between the sequence of ones and the partial product will remain unchanged as the negated MSB will be zero. On the other hand, if the partial product is positive, the negated MSB will be one that will eliminate the entire sequence of ones due to the carry propagation. According to the two's complement representation, this sequence of ones can be represented as $-1 = 111\cdots11$. This process is illustrated in Figure 5.6a.

Once the sign extension is written in terms of -1, it must be further simplified to reduce the number of constant terms. The following steps give the complete sign extension algorithm:

1. Negate the MSB of the signed partial products and add a -1 element on the same weight position. Keep in mind that this step does not introduce any discrepancy to the result.
2. Starting from the least significant bit to the MSB, start removing the -1s according

Figure 5.6: Sign extension optimization on Radix-4 multiplier



(a) Initial step on sign extension for an 8-bit multiplier

(b) Final Sign extension optimization for an 8-bit multiplier

Source: The Author

to rules 3) to 5).

3. If there are two -1 values on weight i, remove these terms and add a -1 on weight $i+1$ since $-1+(-1) = -2$ $(111\cdots10_b)$.
4. If there is only one -1 element on weight i, substitute it for a 1 and propagate a -1 to the next weight.
5. If there are multiple 1, sum them and propagate the carry accordingly to reduce the number of constants and, consequently, the hardware size.
6. Repeat steps 3) to 5) for weight i until all -1 have been cleared.
7. Once weight i is cleared, move to next weight until all positions have been processed.
8. If there are constants $(-1, 1)$ to the left of the MSB, ignore them to avoid useless computation.

The multiplier with optimal sign extension and minimal constant insertion is shown in Figure 5.6b, considering 8-bit inputs. All sign bits are negated (black squares with a top bar), requiring seven additional NOT gates, which can be incorporated directly into the encoders. Finally, the constants can be seen as an additional partial product represented as “0010110”, although the zeros are ignored since they do not convey any information for the circuit.

Table 5.3: Circuit Area and Power Dissipation Comparison at Maximum Speed

Multiplier	8 bits				16 bits				32 bits				64 bits			
	Max Freq. (MHz)	C. Area (μm^2)	T. Power (μW)	E/op (pJ/op)	Max Freq. (MHz)	C. Area (μm^2)	T. Power (μW)	E/op (pJ/op)	Max Freq. (MHz)	C. Area (μm^2)	T. Power (μW)	E/op (pJ/op)	Max Freq. (MHz)	C. Area (μm^2)	T. Power (μW)	E/op (pJ/op)
NR Wallace	344.9	2075.8	645.8	1.9	196.2	6716.3	1066.1	5.4	109.9	23562.8	1938.6	17.6	57.1	78194.5	3353.9	65.6
NR Dadda	357.1	2033.7	674.6	1.9	208.3	6627.9	1108.3	5.3	111.1	20803.6	1831.3	16.5	57.8	75280.9	3376.2	58.4
NR-SO Wallace	344.8	1947.9	638.4	1.8	196.0	6113.1	996.8	5.1	109.9	21002.8	1780.6	16.2	57.8	69218.2	2929.4	50.7
NR-SO Dadda	357.1	1878.2	679.6	1.9	208.3	5975.8	1082.1	5.2	111.1	18092.4	1683.6	15.2	57.8	65187.2	2887.8	50.0
Baseline	302.8	2333.8	929.7	3.1	149.3	8006.4	914.6	6.1	76.9	32765.2	2513.0	32.7	38.3	128500.8	8840.9	230.8

Source: The Author

5.3.2 Performance evaluation

The RTLGen framework was set to generate three versions of the Radix-4 multiplier to assess the benefits of the proposed optimizations: baseline, RCA-less optimization (*NR*) with Dadda and Wallace compression trees and the optimized sign extension without intermediary RCAs (*NR-SO*) architectures considering input widths of 8, 16, 32 and 64 bits. All multipliers employ an RCA as the recombination adder since the optimization focus on the partial product generation and compression. The circuits were synthesized for the ST Microelectronics 65 nm CMOS standard cell library with a power supply of 1.0V using the Cadence Genus synthesis solution with all synthesis efforts set to low for minimal interference. For comparison purposes, all results are obtained at the maximum attainable frequency to extract the circuit quality of results under extreme use cases. The accurate power estimation is guaranteed as it employs the same flow presented in Section 5.2.1.

Table 5.3 summarizes the results considering the maximum attainable frequency (*Max Freq.*), cell area (*C. Area*), total power dissipation (*T. Power*) and energy per operation (*E/op*) for all Radix-4 variants. Both optimization strategies led to circuits that outperformed the baseline version in all cases, and the benefits are more pronounced for larger bit-widths given the longer critical paths due to intermediary adders in the partial product generation and the non-optimal sign extension.

The removal of intermediary ripple-carry adders (version NR) boosts the maximum attainable frequency by a large margin with improvements ranging from 17.9% for 8-bit multipliers, up to 50.9% for 64-bit multipliers. The introduction of the carry-save approach on the partial product compression has a higher impact on the maximum frequency that the sign optimization as the compression tree has a critical path that scales logarithmically. The difference between tree compression algorithms is minimal as Dadda-based multipliers presented a maximum frequency of 2.87% higher on average than their Wallace counterparts.

The optimized multipliers operate at a higher frequency, and they also show remarkably smaller circuit areas at that speed, a direct consequence of better resource utilization. The baseline version requires 24.3%, 33.9%, 81.1%, and 97.1% more area than the best case for multipliers of 8, 16, 32, and 64 bits. The efficiency of compression trees becomes more evident as the operand's bit-width increases. Also, Dadda trees have a smaller circuit area than Wallace trees – 5.6% in our case – as they employ a less greedy algorithm for compressor allocation. Comparing *NR* and *NR-SO* versions, the latter occupies 10.2% less area on average as a reduced number of compressor cells is required due to the smaller number of terms on the tree.

A key point of the improved Radix-4 multiplier is the reduction of circuit power dissipation. The compression tree cells are less prone to timing glitches as the only carry propagation dependency is mainly restricted to the recombination line. Since all circuits were characterized for different clock frequencies, the E/op metric is a more reliable power efficiency indicator. In that sense, the optimized versions require less energy per operation than the baseline implementation with an energy efficiency increase ranging from 16.4% (worst case, 16 bits) to 78.6% (best case, 64 bits). Considering the best case in each bit-width, the average improvement in energy efficiency is 47%. The lower number of elements to be compressed in the *NR-SO* version results in architectures that consume 6.5% less than the version without the sign extension optimization.

5.4 Chapter Summary

The RTLGen framework, presented in this chapter, is a highly flexible, powerful, and versatile Python-based tool that provides an easy platform to describe and generate a plethora of arithmetic circuits along with a unit testbench generated automatically. The modular architecture simplifies the framework extension to support newer architectures. It also features a bit-hash structure that easily manages wire interconnections in multi-operand circuits like multipliers.

The framework was used to generate several multiplier architectures to be synthesized using an industrial ASIC flow. Results showed that commercial synthesis tools fail to natively provide efficient arithmetic circuits in terms of speed, circuit area and power dissipation when compared with RTLGen-based multipliers.

6 CORNET FRAMEWORK: A DEEP LEARNING-BASED SOLUTION FOR HR ESTIMATION

Medicine is evolving from reactive disease care to active care that is predictive, preventive, personalized, and participatory (4P). It provides patients and health care workers with personalized information about each person's unique health experience (FLORES et al., 2013). This paradigm shift has two key benefits: (a) it may anticipate disease detection, which results in better prognosis outcome, and (b) it allows better resource allocation on the health system.

Wearable health-monitoring devices are becoming ubiquitous as advances in both signal processing techniques, and hardware implementation enabled embedding multiple sensors in a single compact and low-power chip. With the evolution of IoT, these sensors can be connected to healthcare networks – providing data of patients with chronic diseases – and social media where users can keep track of their fitness programs (SATIJA; RAMKUMAR; MANIKANDAN, 2017). Further, remote health monitoring reduces healthcare costs and improves resource allocation on health systems (YANG et al., 2018) as it provides data that can be used to accelerate the diagnostic process as well as the treatment selection.

6.1 Heart Rate estimation

Heart rate (HR) monitoring provides essential physiological information, which states the health condition of a given person. Professional and amateur athletes rely on HR measurements to improve their performance on the field as it indicates physiological adaptation, exercise intensity, and workout effort (STRATH et al., 2000). Further, individuals with heart-related diseases may require constant monitoring during daily activities and the sleeping process to help on both the diagnostic process and the selection of the best-suited treatment for optimal outcome (ISLAM et al., 2015).

Wearable sensors have enabled continuous and pervasive vital sign monitoring in ambulant environments, aiding patient care (SESHADRI et al., 2019). In that regard, several companies offer personal health monitoring devices based on smartbands and smartwatches like Fitbit, Apple Watch, among others (KHUSHHAL et al., 2017; DIAZ et al., 2015). Besides HR monitoring, these devices can measure sleep quality, track the fitness

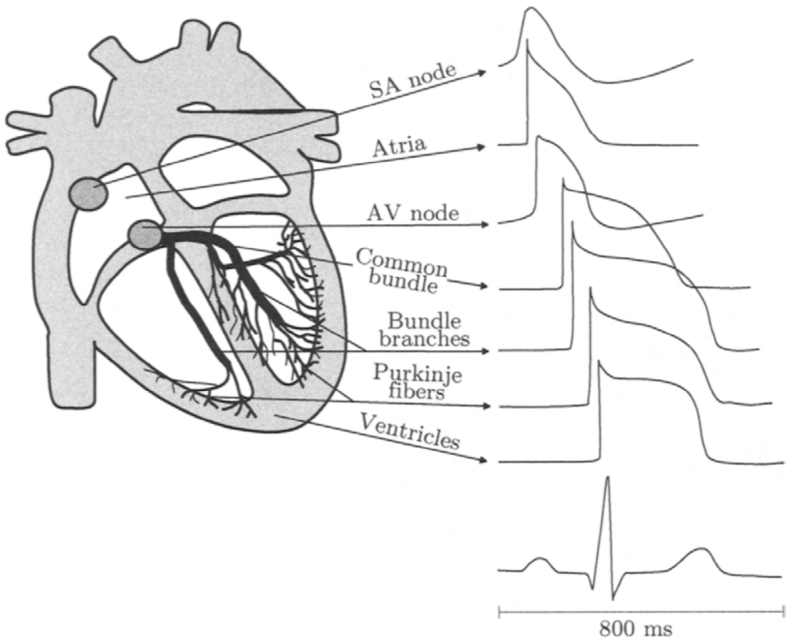
evolution, breathing cycles, and other relevant vital signs. There are multiple techniques to accomplish these measurements, although the most common are based on electrocardiography and photoplethysmography.

6.1.1 Electrocardiogram (ECG)

The sinusoidal cardiac rhythm is due to periodic depolarization and repolarizations of the cardiac muscle, and it is controlled by the sinoatrial (SA) and atrioventricular (AV) pacemaker nodes. This muscle contains several cells that are electrically charged when the muscle is at rest, and when they are stimulated, they depolarize and the cardiac muscle contract (MARTIS; ACHARYA; ADELI, 2014). The electrical impulse propagates throughout the heart and the electric field changes in size and direction.

Electrocardiogram (ECG) signals measure the different electrical phases of the heart excitation, as illustrated in Figure 6.1. The depolarization of each cell group in the heart is an equivalent current dipole source, describing a vector variable in time (SÖRNMO; LAGUNA, 2005). At any given instant, only a group of cells enters this state, generating a current. The ECG will measure the sum of these currents flowing through the cardiac tissue.

Figure 6.1: Electrical potentials of the heart

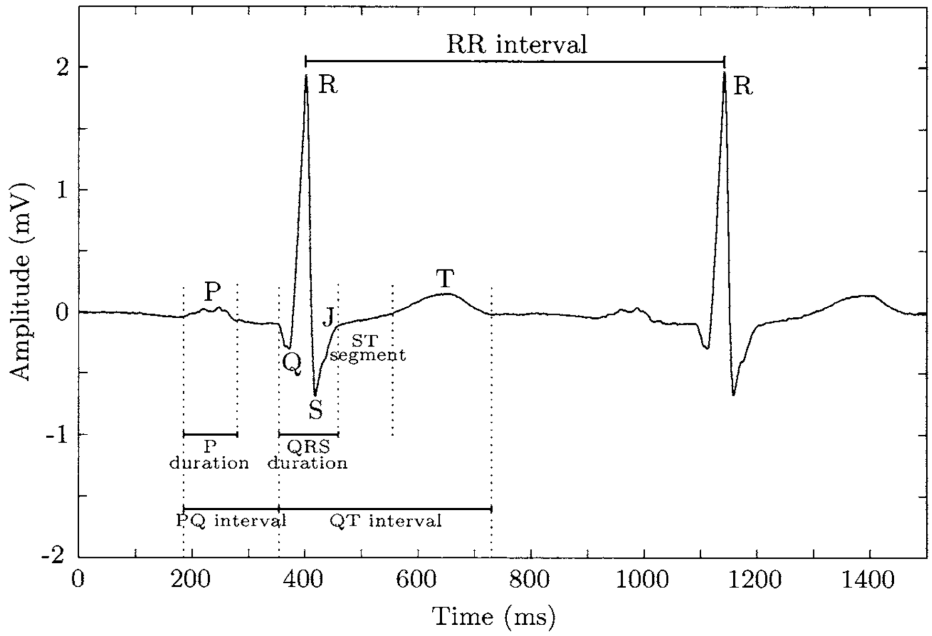


Source: (SÖRNMO; LAGUNA, 2005)

The resulting ECG signal measure different heart phases that dictate the cardiac

rhythm. Figure 6.2 illustrates the P-QRS-T wave phases, where *P* is the atrial depolarization, the *QRS* complex indicate the ventricular depolarization, and *T* reflects the ventricular repolarization (SÖRNMO; LAGUNA, 2005). For heart estimation purposes, the QRS complex is crucial as it features the largest amplitude on the ECG signals. The interval between two R peaks represents the period of a ventricular cardiac cycle, i.e., this period indicates the basal heart rate.

Figure 6.2: Wave definitions of the cardiac cycle



Source: (SÖRNMO; LAGUNA, 2005)

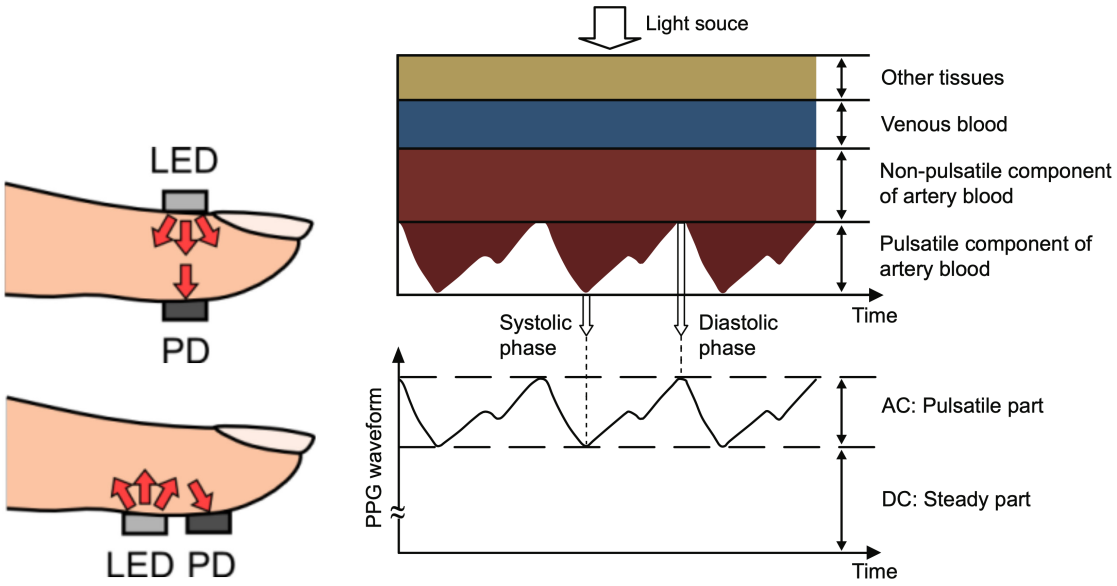
Measuring such low-intensity signals requires several electrodes placed on different places of the chest. Clinical ECG equipment often has 12 leads to measure all the potential vectors of the heart accurately. This approach is robust to motion artifacts that could affect the ECG recording. Nonetheless, the equipment cost and its clunkiness make it impractical for daily usage as a non-stop monitoring system. An alternative version for ambulatory measurements is the Holter device, a 3-lead ECG placed as a chest strap that allows daily activities without being connected to an external device. Despite the robustness of ECG to motion artifacts, they are not exempt from noise sources like bad lead contact or power source interference. Further, these devices do not work when submerged, imposing restrictions on the activity type that can be performed by a given subject.

6.1.2 Photoplethysmography (PPG)

Despite the reliability of ECG measurements to motion artifacts, it requires several probes placed on specific spots on the human body linked to an external processing device, limiting its usability on daily activities (BISWAS et al., 2019a). Photoplethysmogram monitoring is a simple, effective, and low-cost bio-monitoring alternative to ECG, although it can be used to monitor other vital signals. This technique measures the blood volume change, in a non-invasive fashion, on the microvascular tissue under the skin due to the inherent pulsatile characteristic of the cardiovascular system (ALLEN, 2007; AOYAGI; MIYASAKA, 2002).

PPG is based on pulse oxymetry, which relies on a light source – a light-emitting diode (LED) – to illuminate the skin and a photodetector (PD) to capture the attenuation of the reflected or transmitted light associated with perfusion changes. The interaction of the emitted light with the heterogeneous structure of blood and tissues leads to different physical effects – like scattering, absorption, etc. – and it is directly dependent on the emitter optical wavelength (ANDERSON; PARRISH, 1981). Although red and near-infrared lights are used in PPG devices, the green LEDs are predominant because the shorter wavelength is less prone to deep tissue movements, and it results in better readings on the cardiac activity due to its better signal-to-noise ratio (SNR) (BISWAS et al., 2019a).

Figure 6.3: PPG working principle



(a) PPG placement (b) Light attenuation under human tissue

Source: (TAMURA et al., 2014) and (SUN; THAKOR, 2016)

Figure 6.3a illustrates how the LED and PD can be placed on the body for PPG

measurements. On the transmissive mode (top), the LED and the photodetector are placed on opposite sides, so the cardiac activity is measured in terms of light attenuation. Although this approach often has better SNR, it has a severe limitation in terms of placement because the light must be able to pass through the tissue like earlobes, fingertips, and so on, which may interfere with daily activities (TAMURA et al., 2014). Hence, reflective mode (bottom) is preferred as both LED and PD are side-by-side, effectively removing all sensor placement restrictions. In this mode, the LED illuminates the skin while the PD measures how much light is reflected by the tissue. Nevertheless, reflection-mode PPG is significantly affected by motion artifacts and sensor pressure disturbances, limiting the physiological interpretation of the captured signals (BISWAS et al., 2019a).

The working principle of PPG sensors is illustrated in Figure 6.3b. As the light penetrates the tissues down to the blood vessels, the diastolic and systolic movements create variations on the transmitted/reflected light captured by the photodetector. The detected signal has a steady (DC) and a pulsatile (AC) components. The former depends on the tissue thickness and composition, absorbance by the skin pigmentation, blood volume, and it slowly changes according to respiration. The AC component captures the blood volume changes during the cardiac cycles, whereas the main frequency component is directly related to the heart rate. Further, the AC signal amplitude depends on the contact force between the sensor patch and the skin (TAMURA et al., 2014).

Most wrist-based modules employ PPG sensors allowing unobtrusive daily usage. However, the signal quality in these devices is affected due to motion artifacts (MA), resulting from activities of daily living. MA are generally due to three reasons: the subject physical movement, sensor module displacement relative to the skin, and sensor deformation resulting from long-term daily usage. These effects cause the spectral component of the MA to overpower the heart-beat related PPG component.

6.1.3 Public datasets for HR estimation

Significant advances in wrist-worn PPG research was due to the availability of public datasets with specific protocols. For the Signal Processing Cup (SPC) fostered by the IEEE in 2015, the authors in Zhang, Pi and Liu (2015) made public a dataset containing 23 records of healthy subjects with age ranging from 18 to 58 years old. All recordings were captured using a 2-channel pulse oximeter with green LED (wavelength: 515 nm) along with a 3-axis accelerometer to measure the subject's motion. To capture

the ground-truth HR, all subjects wore a chest-worn ECG with wet electrodes. The raw data were recorded with a sampling rate of 125 Hz and transmitted to a computer using Bluetooth. This dataset is commonly known as the IEEE SPC dataset.

The subjects performed three different types of activities, namely T1, T2, and T3. The first 12 subjects performed activity T1 which consists in a protocol that involves walking and running on a treadmill with the following speeds and duration, in order: 1-2 km/h for 0.5 min, 6-8 km/h for 1 min, 12-15 km/h for 1 min, 6-8 km/h for 1 min, 12-15 km/h for 1 min, and 1-2 km/h for 0.5 min. This protocol explores the correlation between past and current body states as the heart rate on protocol start is entirely different from the one in the end, even though the subject is walking in both situations.

Table 6.1: IEEE SPC dataset overview

Subjects	Age (years)	Weight (kg)	Height (cm)	Gender (M/F)	Healthy	Activity (T1/T2/T3)
1-12	18-35	N/A	N/A	N/A	Healthy	T1
13	20	64	162	M	Healthy	T2
14	29	70	169	M	Healthy	T2
15	21	77	188	M	Healthy	T2
16	21	77	188	M	Healthy	T3
17	19	54	174	M	Healthy	T3
18	20	64	162	M	Healthy	T3
19	20	57	174	M	Healthy	T3
20	19	70	180	M	Healthy	T2
21	19	70	180	M	Healthy	T3
22	21	73	180	M	Healthy	T3
23	58	70	156	F	Abnormal	T2

Source: Adapted from (CHUNG; LEE; LEE, 2019)

Records 13, 14, 15, 20, and 23 performed T2 activity, which involved running, push-ups, handshaking, stretching, and other upper arm movements. The remaining subjects (16-19, 21, 22) executed activities T3, which consisted of intensive fore and upper arm movements like boxing, which are activities very correlated with intense motion artifacts in PPG signals. Table 6.1 shows an overview of the IEEE SPC dataset. Several works ignore record 13 as it is considered an extra dataset, and it was made available later than the other records.

Other works also developed publicly available custom datasets, even though they are not extensively explored in the literature. In Jarchi and Casson (2016), the authors cre-

ated a dataset with nine records with the same subject performing four activities: walking and running on a treadmill, bike exercise with low resistance/high speed, and bike exercise with high resistance/low speed. The data was captured using a wrist-worn PPG using a green LED (wavelength: 510 nm), a 3-axis accelerometer, and a 3-axis gyroscope for motion estimation and a chest-worn ECG sensor for the ground-truth. Another dataset was proposed in Lee, Chung and Lee (2019), and it included 24 subjects (10 males, 14 females), which performed a modified version of the Bruce protocol on a treadmill in five stints that interleaved 2 minutes of walking with 3 minutes of running. The records were captured using a 3-channel wrist PPG with green LED (wavelength: 525 nm) along with a 3-axis accelerometer and gyroscope sensors. For the ground-truth heart rate, the authors used a Holter device to capture the ECG data.

6.2 CorNET framework for HR estimation

The CorNET framework proposed by Biswas et al. (2019b) was the first machine learning-based solution for HR estimation from PPG signals on ambulant environments. It combines the advantages of feed-forward CNNs as feature extractors and the recurrent nature of LSTMs suited for time series handling. The framework relies on ECG-generated HR measurements as ground truth during the training phase, learning the relationship between each PPG window and the HR computed from the corresponding ECG frame.

Once trained, the network can estimate the HR from single-channel PPG measurements without any additional data like accelerometers and additional PPG channels. The framework adopts a subject-specific (personalized) training methodology instead of a subject-independent (generalized) approach based on the fact that biological signals are heavily dependent on physiological aspects like age, sex, weight, physical activity habits, and so on. However, the work in Rocha et al. (2019) proposes a subject-independent training methodology based on two premises: (a) it is more feasible for deployment in embedded applications due to faster training times, and (b) training on multiple subjects may improve the model ability to incorporate the inherent physiological variations on the human body. As an additional experiment, the framework was evaluated on a custom dataset obtained with specific wrist-worn smartband able to capture PPG data along with an off-the-shelf chest-worn ECG monitoring band.

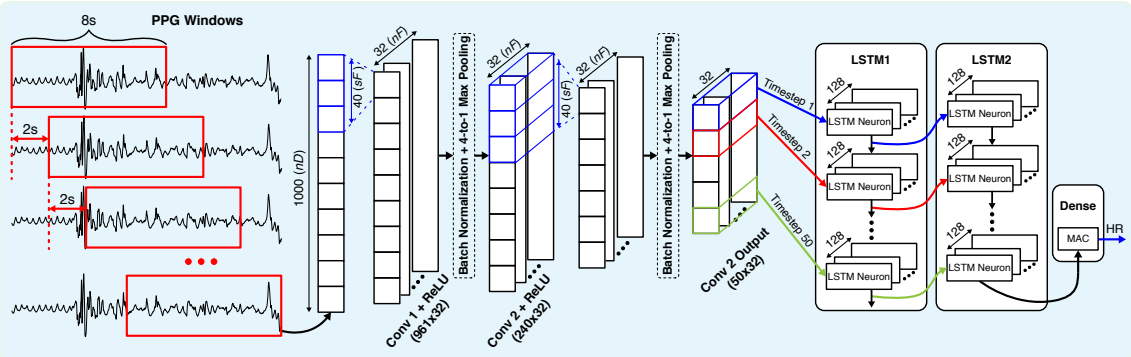
6.2.1 Network structure

Conversely to the popular image classification neural networks that translate pixel images into discrete categories, the CorNET model must infer a continuous HR value from voltage values that might include motion artifacts that corrupt the PPG signal, lowering the model accuracy. This issue is addressed using a data-driven approach that allows the neural network to learn unique features using convolutional layers and combine the temporal dependencies with recurrent layers to generate an output vector which can be combined with a dense layer (DL) for the final HR estimation.

The CorNET network employs two 1-D convolutional layers, which act as a feature extractor of the incoming data. The convolution between the CNN filters and the input PPG signal generate points on the temporal-feature domain. Each CNN layer has 32 filters (nF), and each filter has 40 coefficients (sF) operating with a stride of one sample. Both layers use ReLU as activation functions whose outputs are connected to a batch normalization and 4-to-1 max-pooling layers.

A major problem in traditional CNNs is that they cannot guarantee that the output is phase invariant, requiring complementary layers that perform time pooling to mitigate this issue (PURWINS et al., 2019). In HR estimation, this is particularly troublesome as the precise moment of a given heartbeat within a PPG window is fundamental to correctly identifying the relevant features linked to the estimated heart rate.

Figure 6.4: CorNET architecture operating on 1-D input samples to predict HR



Source: The Author

The ability to capture temporal dependencies of the LSTMs makes them a suitable choice to solve the phase variance issue as they can recover the sequence of local trends caused by the cardiac activity. Hence, the CorNET network has two sequentially connected LSTM layers after the CNN layers, whereas each layer has 128 hidden units. The hidden state of the last LSTM is fed to a regression layer with a single neuron with 128

connections, which is followed by a linear transfer function to estimate a real-valued heart rate. The overall network structure can be seen in Figure 6.4. All the structural parameters were defined following a heuristic grid search, which looked for the minimal number of parameters that yielded the best HR estimation performance (BISWAS et al., 2019b). Table 6.2 summarizes the number of parameters in the model as well as the number of operations required per output inference.

Table 6.2: Complexity Evaluation of CorNET

Layer	**Trainable Parameters**	**MACs**	**Memory Requirement (bytes)**
CNN-1	1312	1.2 M	5248
CNN-2	40992	8.2 M	163968
LSTM-1	82432	4.1 M	329728
LSTM-2	131584	6.6 M	526336
Dense	128	128	512
Total	256448	∼20.1 M	∼1000k

6.2.2 Training methodology and evaluation

The CorNET framework for HR estimation was modeled on the Lasagne 0.2dev1 (DIELEMAN et al., 2015) library configured to use Theano (THEANO DEVELOPMENT TEAM, 2016) 0.9.0 as the computation backend. The training procedure was executed on an Nvidia GTX 1080 GPU with cuDNN 7.6 for faster training time (CHETLUR et al., 2014). For the parameters update, the original work in Biswas et al. (2019b) adopts the Root Mean Square Propagation (RMSProp) optimizer with default hyperparameters while the work in Rocha et al. (2020) adopts the Adam optimizer with default parameters for the non-binary framework implementation.

There are two training methodologies for the CorNET framework. The *leave-one-window-out* (LOWO) strategy is a personalized method (subject-specific) based on the premise that each subject has specific physiological factors. In this method, for a given subject, each window is removed from the data pool and used as the testing dataset, while the remaining ones are used as the training dataset. Further, three windows on the training dataset adjacent to the testing window are removed to ensure that there is no data overlap between training and testing datasets as the windows have a 6 seconds overlap.

In Rocha et al. (2019), the authors propose a *leave-one-subject-out* (LOSO) strat-

egy, a generalized method that assumes that training in a broader population enables the model to more likely integrate the physiological variations into a single model. The benefits of this strategy are twofold: (a) it is easier to be deployed on a large scale as it does not require a large amount of data for each specific subject, and (b) it might help to identify if the subject has any type of cardiopathy. This method relies on removing the entire data of the subject under test (SUT) from the global dataset while the remaining subject records form the training dataset, ensuring that there is no overlap between training and testing data. Once the model is trained, the data from the SUT is used to evaluate the model capability to generalize for unseen data.

All training methods are validated using a 5-fold cross-validation method on the training dataset, and the model with the lowest validation error is chosen. Each fold is trained for 200 epochs, considering a batch size of 25 and 32 for LOWO and LOSO, respectively, with an initial learning rate of 0.001 associated with a decaying factor of 0.98 after each epoch.

Regardless of the target dataset, the input PPG data is pre-filtered with a 4^{th} order band-pass Butterworth filter with cut-off frequencies set to 0.2 Hz and 4 Hz to eliminate DC/near-DC components and frequency noises. These frequencies are selected in terms of the heart's pulsatile characteristic, limiting the estimated heart rates to 12 BPM and 240 BPM, respectively. Although these frequencies are outside the typical physiological limits of the heart, they ensure a safety margin on input data. Further, a z-score normalization (zero mean, unit variance) is applied to the inputs as the class labels (HR) do not have a normal distribution, and smaller input values improve the training performance (STÖTTNER, 2019).

The metrics adopted to evaluate the model performance are the mean absolute error (MAE) and the standard deviation of the absolute error (SDAE) of the estimated heart rate in beats per minute (BPM). These are the standard metrics on the PPG-based HR estimation state-of-the-art (ZHANG; PI; LIU, 2015). The MAE is computed in a window-by-window basis considering the HR estimated by the CorNET model (HR_E^i) and its ground-truth counterpart HR computed offline from the ECG data (HR_T^i), as stated in (6.1), where N is the number of PPG windows for a given subject.

$$\text{MAE} = \frac{1}{N}\sum_{i=0}^{N-1} \text{abs}(\text{HR}_E^i - \text{HR}_T^i) \tag{6.1}$$

The overall performance evaluation of the CorNET network using both LOWO

and LOSO training strategies is presented in Table 6.3 considering the IEEE SPC dataset. It also compares the CorNET performance against the HR estimation from PPG data proposed in Troika (ZHANG; PI; LIU, 2015), Joss (ZHANG, 2015), WFPV (TEMKO, 2017), and Chung (CHUNG; LEE; LEE, 2019). These works use signal processing techniques instead of machine learning methods and rely on additional accelerometer data to reduce motion artifacts. The analysis considers the MAE for each dataset record and each group of subjects performing a specific activity (T1, T2, and T3). Note that only the WFPV and Chung methodologies consider the 23 records on the IEEE SPC dataset because record 13 was considered extra training data since this dataset became public. Therefore, the error reported for all subjects (1-23) on these two methodologies includes this specific record. Yet, this single record does not significantly impact the comparison of the HR estimation approaches.

As expected, all methodologies report the smallest MAE on the subjects performing activity T1 since they follow a very specific protocol. This is particularly relevant for CorNET due to an increased amount of similar data on the training dataset, which is directly correlated with the model's ability to capture the relevant characteristics of the input signal. Despite using only the PPG data, the CorNET with the LOWO strategy has comparable performance to other works. Compared to Troika and Joss, the LOWO approach has a 27.84% and 5.28% lower MAE, respectively, considering all subjects, and it is only outperformed by the Chung algorithm.

Nevertheless, the overall performance of the LOSO approach is not optimal when compared to its counterparts. Compared to the LOWO approach, it has an MAE 2.8$\times$ higher considering all records. This worse performance is mainly due to the limited number of records on the dataset, as the LOSO methodology aims for a generalized approach. For instance, the work in (ROCHA et al., 2019) reports the accuracy of the LOSO approach for only the first 12 subjects, achieving an absolute error of 3.16$\pm$6.02 BPM. When the full dataset is employed on the training process, the reported absolute error is 32.32% higher than when the dataset is limited to only the subjects performing the same activity.

For a more in-depth analysis, subjects 9 and 17 were selected to illustrate the HR estimation performance of the CorNET LOSO as they represent the best and worst subjects, respectively. Figure 6.5 shows the predicted (blue) and ground-truth (orange) HR for record 9, indicating the current subject running speed in each time interval. The CorNET framework can closely follow the HR changes across the activity duration. Note

Table 6.3: Performance evaluation of HR estimation algorithms and CorNET

Record	Troika	Joss	WFPV	Chung	CorNET (LOWO)	CorNET (LOSO)
1	2.29±2.18	1.33± 1.19	1.25± 1.15	0.73±0.59	6.23± 9.44	4.26 ± 3.01
2	2.19±2.37	1.75± 1.66	1.41± 1.30	0.81±0.68	1.83± 5.18	9.10 ± 7.52
3	2.00±1.50	1.47± 1.27	0.71± 0.59	0.54±0.41	0.89± 3.49	3.79 ± 3.31
4	2.15±2.00	1.48± 1.41	0.97± 0.88	0.72±0.58	0.49± 2.29	3.22 ± 3.13
5	2.01±1.22	0.69± 0.51	0.75± 0.57	0.59±0.43	0.40± 1.01	2.35 ± 1.82
6	2.76±2.51	1.32± 1.09	0.92± 0.75	0.87±0.69	3.08± 6.47	2.76 ± 1.98
7	1.67±1.27	0.71± 0.54	0.65± 0.50	0.66±0.52	1.34± 4.42	2.18 ± 1.36
8	1.93±1.47	0.56± 0.47	0.97± 0.83	0.63±0.52	3.64±10.19	10.84 ± 8.68
9	1.86±1.28	0.49± 0.41	0.55± 0.48	0.43±0.35	3.30± 6.81	1.68 ± 1.04
10	4.70±2.49	3.81± 2.43	2.06± 1.29	1.44±0.98	1.77± 3.96	8.49 ± 7.06
11	1.72±1.29	0.78± 0.51	1.03± 0.68	1.14±0.73	0.41± 1.37	3.22 ± 2.19
12	2.84±2.30	1.04± 0.81	0.99± 0.70	0.96±0.68	0.50± 1.05	4.14 ± 3.40
13	-	-	3.54± 4.08	2.51±2.85	-	-
14	6.63±8.76	8.07±10.09	9.59±12.20	0.60±0.99	1.60± 2.29	9.67 ± 8.75
15	1.94±2.56	1.61± 2.01	2.57± 3.16	0.91±1.34	0.24± 0.56	4.35 ± 3.44
16	1.35±1.04	3.10± 2.69	2.25± 1.87	1.04±0.82	1.60± 3.87	7.03 ± 6.96
17	7.82±4.88	7.01± 4.49	3.01± 1.99	1.72±1.29	2.04± 5.02	12.62 ± 9.15
18	2.46±2.00	2.99± 2.52	2.73± 2.29	1.07±0.87	0.95± 3.02	5.41 ± 4.63
19	1.73±1.28	1.67± 1.23	1.57± 1.15	0.90±0.67	0.28± 0.60	4.31 ± 2.79
20	3.33±3.90	2.80± 3.46	2.10± 2.41	1.32±1.59	0.28± 0.65	5.28 ± 3.54
21	3.41±2.43	1.88± 1.32	3.44± 2.45	1.23±0.89	0.67± 1.09	11.45 ± 9.27
22	2.69±2.12	0.92± 0.74	1.61± 1.26	1.35±1.07	0.42± 0.73	3.67 ± 2.79
23	0.51±0.59	0.49± 0.57	0.75± 0.88	0.69±0.82	0.75± 0.88	2.35 ± 2.16
Records 1-12 (T1)						
MAE	2.34±2.47	1.28±2.61	1.02±1.25	0.79±0.60	1.99±4.64	4.67 ± 3.71
Records 14-23 (T2 and T3)						
MAE	3.19±3.61	3.05±3.35	2.95±3.71	1.07±1.02	2.95±3.71	6.61 ± 5.35
Records 1-23 (T1, T2 and T3)						
MAE	2.73 ± 2.99	2.08±1.91	1.97±2.48	0.99±0.88	1.97±2.48	5.55 ± 4.45

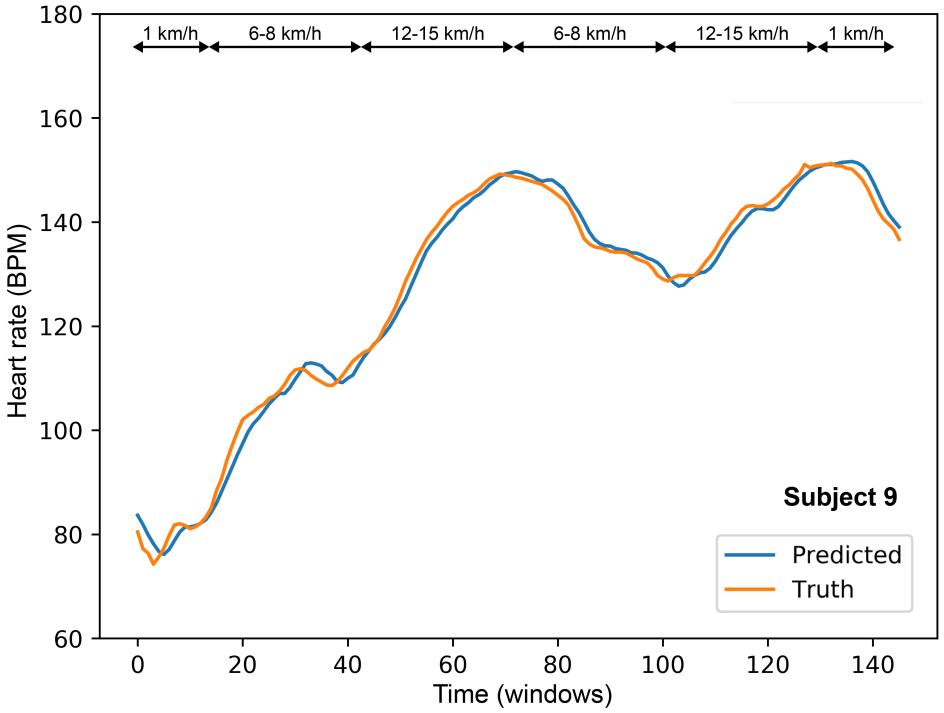
Source: The Author

that this subject has a steady and progressive HR increase without significant peaks in a short period.

Conversely, Figure 6.6 shows the HR estimation analysis for subject 17. The quickly and acute HR rise associated with the activity type (intense upper-arm exercises) creates severe challenges for the model as it cannot find similar patterns on the data of other subjects of the training dataset. Since the data is captured from a wrist-worn PPG sensor, the intense motion artifacts cannot be easily removed when no additional data

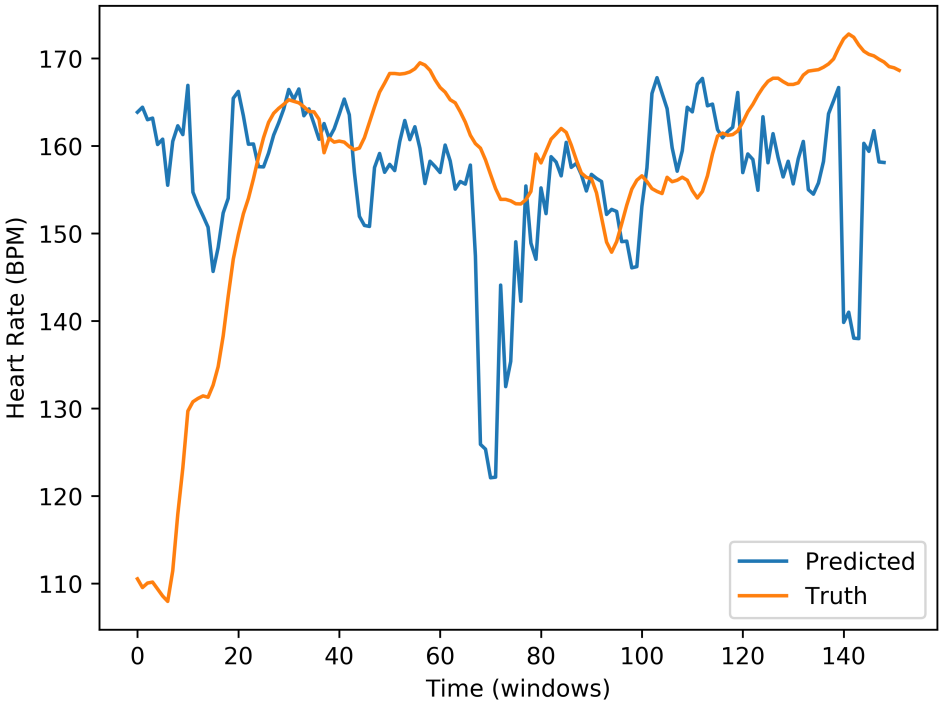
is used, leading to huge variations on the predicted HR that might not correspond to a plausible physiological change.

Figure 6.5: PPG Estimated vs true ECG HR for Subject 9



Source: The Author

Figure 6.6: PPG Estimated vs true ECG HR for Subject 17



Source: (ROCHA et al., 2020)

6.3 Chapter Summary

This Chapter presented a brief overview of heart rate estimation from both ECG and PPG sensors. This biomedical application fostered the development of several algorithms to achieve real-time HR estimation capability from PPG data due to its low cost, portability and non-intrusive characteristic.

CorNET was one of the first deep learning-based implementation targeting this biomedical application. It combines CNNs and LSTMs to explore the feature extraction characteristic of the former with the inherent ability of capturing temporal dependencies of the latter. Results showed that CorNET is in pair with other state-of-the-art HR estimation algorithm based on traditional signal processing techniques.

7 STREAM-BASED HARDWARE IMPLEMENTATION FOR BINARY CORNET FRAMEWORK

Computing- and power-efficiency are crucial in wearable devices as they have limited resources – especially power delivery – given the small form factor and usability requirements. Despite the efficiency of modern embedded general-purpose processors, they do not comply with the requirements to execute efficiently compute-intensive applications like neural networks. Hence, a custom hardware implementation is required for the binary CorNET (bCorNET) neural network.

7.1 CorNET model modifications

The CorNET framework is not adapted for embedded platforms as initially proposed in Biswas et al. (2019b) since it is a pure software implementation that relies on floating-point computations without any processing constraints like energy consumption, computation capacity available, and so on. Hence, two essential modifications were proposed in Rocha et al. (2019) to adapt the original framework into a low-power, hardware-suited implementation through input data quantization and weight binarization.

7.1.1 Data quantization

The floating-point representation adopted in available PPG databases is not suitable for real-time embedded devices due to the huge complexity. Hence, these devices must adopt data quantization to reduce the complexity requirements. This strategy has two main benefits: first, the arithmetic modules are considerably simpler as they can adopt a fixed-point implementation, and it may help on overfitting in neural networks due to the induced quantization error. It is essential to determine the optimal value range as it impacts both network accuracy and the input bit-width. This issue is addressed using a learning-based quantization approach that seeks, during training, the best input range to represent the incoming data. The proposed quantization method follows a uniform quantization rule:

$$quantize(x) = round(clip(x, -1, 1) \times M)/M \tag{7.1}$$

$$Q(x) = s \times quantize\left(\frac{x-p}{s}\right) + p \tag{7.2}$$

In these equations, x is the network input, and M is the number of strictly positive quantization levels, which is given by $M = 2^{(nb-1)} - 1$. Following a previous exploration, nb was set to 5 bits. Further, the quantization has two additional parameters learned during the network training: s is a scaling factor, and p is an offset parameter that helps determine the optimal quantization range. This strategy limits the number of different input levels to M, and these values are specific to each model.

7.1.2 Model binarization

Neural network models are defined through thousands of parameters that are naturally defined using floating-point representation, and these models require thousands of operations per valid output. This complexity can be addressed through network binarization to reduce both memory requirements and constraints on the hardware implementation. Model binarization was introduced on the BinaryConnect network (COURBARIAUX; BENGIO; DAVID, 2015), and it has been proven as an efficient technique on CNN networks to reduce model complexity (COURBARIAUX; BENGIO, 2016; RASTEGARI et al., 2016).

Despite the several binarization strategies available on the literature, both CNN and LSTM layers on bCorNET are binarized using the error-aware binarization technique proposed in Hou, Yao and Kwok (2016). In this approach, the error introduced by the binarization is considered on the training process, and the weights are updated accordingly to minimize the training loss. The full-precision weights are used during the update phase, while the inference phase is computed using the binarized values. Here, the parameters that belong to the same weight set (e.g., $W_* = [0.325, -0.325 \cdots -0.325]$) have the same magnitude, although the sign may differ. The magnitude is factored out, and it can be interpreted as a scaling factor λ_* associated with each weight set W_* that can be rewritten as $W_* = 0.325 \times [1, -1 \cdots -1]$. Each weight set can be stored in terms of a fixed-point scaling factor and a binary sequence corresponding to each value sign.

Each binary CNN (bCNN) layer has a single scaling factor shared with all filters

belonging to that layer. Then, each element x on the output tensor of filter $k \in [0, nF-1]$ in a given layer is computed according to (7.3), where nF is the number of filters in that layer and sF is the number of coefficients in each filter. Since bCorNET adopts the same structural parameters defined for the CorNET model, both nF and sF are set to 40.

$$o_{kx} = bin\left(ReLU\left(\lambda^q_{Wc}\sum_{c=0}^{nC-1}\sum_{i=0}^{sF-1} w^b_{ci} I_{i+x}\right)\right) \tag{7.3}$$

Each output is computed multiplying the filter binary coefficients (w_{ci}) with a slice of the input feature map I composed of sF values, starting at position x. The bCNN1 inputs are quantized to 5 bits, while bCNN2 operates with both inputs and weights in a binarized fashion. Further, λ^q_{Wc} is a scaling factor derived from the binarization process, which is specific to each layer. Since the incoming PPG data is 1-dimensional, the Equation (7.3) is reduced to only the second sum operation due to the single input channel. Conversely, the second bCNN layer has nF input channels as it is sequentially connected to bCNN1.

The binarization on the LSTM (bLSTM) layers is slightly different as there are internal computations that cannot be entirely binarized without losing the layer functionality. Each layer is comprised of an input gate (i_t), a forget gate (f_t), an output gate (o_t) and a cell state gate (u_t). These gates compute intermediary values from the current input (x_t) and the previous hidden state (h_{t-1}) as follows:

$$i^q_t = \sigma(\lambda^q_{Wi}(W^b_i x^b_t) + \lambda^q_{Ui}(U^b_i h^b_{t-1})) \tag{7.4}$$

$$f^q_t = \sigma(\lambda^q_{Wf}(W^b_f x^b_t) + \lambda^q_{Uf}(U^b_f h^b_{t-1})) \tag{7.5}$$

$$o^q_t = \sigma(\lambda^q_{Wo}(W^b_o x^b_o) + \lambda^q_{Uo}(U^b_o h^b_{t-1})) \tag{7.6}$$

$$u^q_t = \tanh(\lambda^q_{Wu}(W^b_u x^b_t) + \lambda^q_{Uu}(U^b_u h^b_{t-1})) \tag{7.7}$$

$$c^q_t = f^q_t \times c^q_{t-1} + i^q_t \times u^q_t \tag{7.8}$$

$$h^b_t = sign(o^q_t \times \tanh(c^q_t)) \tag{7.9}$$

These equations assume that x_t is the input to a given LSTM layer, W_* and U_* are weight matrices, and h_t is the output of the LSTM. From the LSTM definition, the h_t and c_t values become the h_{t-1} and c_{t-1} values, respectively, on the next timestep. Further, the superscripts b and q indicate binarized and quantized values, respectively. The number of bits for all quantized values is determined by the LUT implementation parameters of the

non-linear functions, described on Section 7.2.5.

The binarization strategy ensures that MAC operations are performed with binarized operands (weights and inputs), except for the first bCNN layer whose inputs are quantized accordingly. Since most of the weights are represented using a single bit, the network's memory footprint is reduced by a factor of 32, approximately, compared to the standard 32-bit floating-point representation of the software model. The identical structure between CorNET and bCorNET ensures that the number of MAC operations and trainable parameters remains constant, yet with two key advantages: (a) the MAC operations are reduced to XNOR operations based on a popcount algorithm, and (b) the memory requirements are reduced from $\sim$1000 kbytes to $\sim$31 kbytes.

7.1.3 Training optimization for model binarization

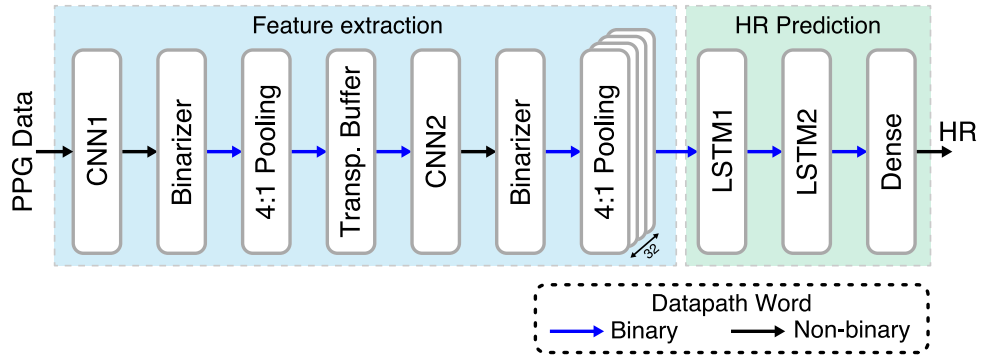
Model binarization leads to non-differentiable weight update functions, which breaks the ground premise of the backpropagation algorithm. In bCorNET, this issue is addressed using the straight-through estimator (STE) proposed in Courbariaux and Bengio (2016). Further, the loss function optimizer is a modified version of the Adam algorithm. This binarization-aware optimizer was proposed in Hou, Yao and Kwok (2016), and it relies on an additional step to compute the current curvature matrix that leads to the smallest loss value, which is used to update the full-precision weight parameters. A key element in this process is using the binarized values during the feed-forward step while keeping the original full-precision parameters for the best training outcome.

7.2 System architecture

Hardware accelerators for CNN-based neural networks have been widely explored in the literature, as explored in Section 3.4.3. However, very few works address the implementation challenges of LSTM-based networks, given their recurrent nature and high memory requirements. The first move towards a custom hardware implementation for the bCorNET network was proposed in (ROCHA et al., 2019), introducing an FPGA-based of binary LSTM layers. A follow-up of this work was proposed in Rocha et al. (2020), which introduced a stream-based hardware architecture for the bCorNET network, which fully integrates the binary CNN and LSTM layers along with the quantized dense layer.

The proposed hardware architecture top-level for HR estimation is illustrated in Figure 7.1. Each layer on the network model is translated into a dedicated block on the hardware architecture in a trade-off between flexibility and energy efficiency.

Figure 7.1: bCorNET hardware accelerator architecture



Source: The Author

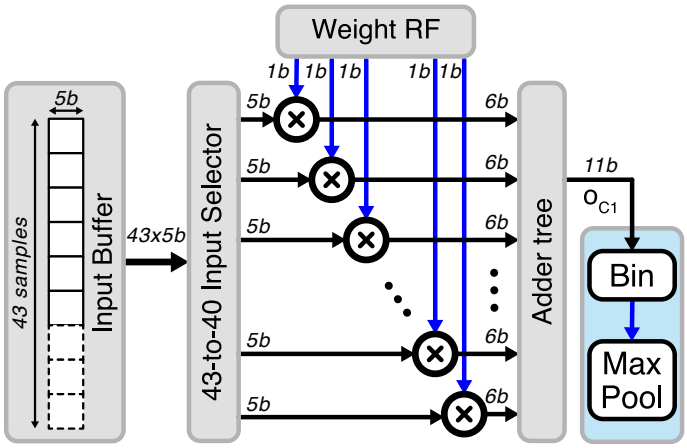
The architecture relies on a pipelined approach for processing the entire network in a streaming fashion, saving both memory requirements and system latency. As each layer has data ready at the output, it sends a signal to the system controller, which activates the subsequent layer and enables the previous layer to fetch new data. A non-stream version of this architecture would require near 100 kbits of additional memory for intermediary storage, and it would take three times longer to process a PPG window entirely. This design choice is suited for embedded circuits in wearables as it leads to smaller circuit footprint and operating frequency, two fundamental aspects to reduce power dissipation (KILANI et al., 2020). Given the identical network architecture of CorNET and bCorNET, the number of MAC operations and trainable parameters remain unchanged, however: (a) all the MAC computations are reduced to XNOR operations plus popcount, and (b) a considerable reduction is achieved in the memory footprint for weight storage, since the latter requires only $\sim$31 kbytes instead of $\sim$1000 kbytes.

For optimal resource utilization, the datapath width is not constant for all blocks. The incoming PPG data is quantized to 5 bits according to the network definition defined in Section 7.1.1, while each layer has a custom data width to comply with specific requirements. For instance, CNN and LSTM layers have intrinsic architecture differences – like using non-linear functions – which entails different data representation constraints. Further, all arithmetic operators used on the hardware blocks were generated using the RTLGen framework presented in Chapter 5. Since the main goal is obtaining an energy-efficient hardware architecture, circuit-level low-power techniques like data- and clock-gating were employed in all suitable modules.

7.2.1 CNN1 Layer

The first convolutional layer handles the incoming 1-dimensional PPG and, given that each window has 1000 samples and the stream-like dataflow aspect of the network, this module adopts a weight-parallel implementation with input data reuse to minimize data movement. Figure 7.2 shows the proposed architecture for this layer, composed of an input buffer, a register file (RF) for weight storage, an array of multipliers, and an adder tree.

Figure 7.2: CNN1 layer architecture



Source: (ROCHA et al., 2020)

Each filter computation is composed of 40 multiply-accumulate operations, and it is completed in a single cycle due to complete parallelization. Despite the model binarization, this module operates in a hybrid fashion with quantized inputs and binary weights, so the multiplier is reduced to a multiplexer to select between the data value and its 2's complement version. Each multiplier output is 6-bit wide to ensure that the 5-bit incoming PPG data does not overflow when computing the 2's complement. The adder tree operates in a carry-save scheme with a synthesis tool-inferred adder on the recombination line, so the layer output is 11 bits wide to ensure that no overflow occurs. The layer output is then connected to a binarizer module before the max-pooling.

Keeping data local for computation is proven to be more energy-efficient as the energy cost for each data access on external memories is humongous (VERHELST; MOONS, 2017). Hence, the input buffer receives the data sequentially from an external data source (e.g., analog front-end, memory, etc.) and it has two roles: (a) control the communication between the CNN and external modules, and (b) offer a quick and efficient way for the CNN1 to access data. It features a register-based FIFO memory able to store up to

43 words and capable of serial write (from external source) and parallel read (by filter multipliers).

Since each CNN1 filter has 40 coefficients and the layer adopts a sliding window with a stride of 1, each loaded filter can be reused four times, considering the input buffer storage capacity. This approach requires the usage of multiplexers on the buffer output to select 40 words out of the 43 available. Reusing each filter four times results in four sequential outputs for each filter, so no intermediary memory is required to store the binarized CNN1 results before the 4:1 max-pooling kernel.

The input buffer external interface is 20-bit wide, corresponding to 4 words of 5 bits per reading request. When a new PPG window is available, the controller starts fetching new data every cycle, filling the buffer in 11 cycles. Note that in the last cycle, only three words are loaded into the buffer, and the first filter coefficients are loaded into the multipliers, starting the window processing. Due to input reuse, a new set of filter coefficients is loaded every 4 cycles, and a subsequent input data fetches occur after all the 32 filters have been multiplied by the current input set, i.e., the input buffer loads 4 new data words into the highest addresses after $4 \times 32 = 128$ cycles.

7.2.2 Binarizer and Max-pooling layers

Each convolutional layer is followed by a ReLU activation function, a batch normalization layer, and a hard tanh function used to binarize the data for the next layer. In this case, the ReLU eliminates all negative values and sends them to the BN layer, which applies a scale-and-shift operation to remove the covariate shift. Then, the binarization function applies a hard threshold comparison with the threshold set to zero to determine if the output must be $+1$ or -1 (corresponding to 0 and 1 in binary representation respectively). The batch normalization output (o_{BN}) is computed according to (7.10), where μ and σ are the batch mean and variance, respectively, γ and β are the linear scale and shift parameters and ϵ is a numeric stability constant set to 10^{-4} (IOFFE; SZEGEDY, 2015; DIELEMAN et al., 2015). Given the weight binarization method employed on the convolutional layer, the BN layer input can be expressed in terms of αo_C, where o_C is the accumulated value on the CNN layer and α is the scaling factor associated with the filters.

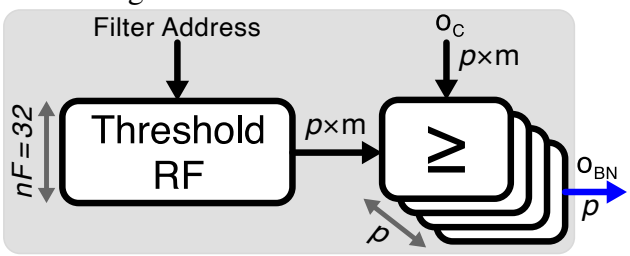
$$o_{BN} = \frac{\alpha o_C - \mu}{\sqrt{\sigma^2 + \epsilon}}\gamma + \beta \tag{7.10}$$

The binarization function must detect if the o_{BN} value is positive or negative and set the output accordingly ($+1$ or -1). Then, rewriting (7.10) shows that the o_{BN} value will only be positive if the CNN value (o_C) is higher than a threshold whose value is derived from parameters learned during training, as shown in (7.11). The ReLU activation can be ignored as the shift parameter on the batch norm layer will ensure that only positive o_C values can be mapped to $+1$, i.e., all negative and some positive o_C values will be mapped to -1.

$$\frac{\alpha o_C - \mu}{\sqrt{\sigma^2 + \epsilon}}\gamma + \beta \geq 0 \quad \Longrightarrow \quad o_C \geq \frac{\mu}{\alpha} - \frac{\beta\sqrt{\sigma^2 + \epsilon}}{\gamma\alpha} \tag{7.11}$$

Each CNN layer has specific output range requirements since the CNN1 operates in single-channel input while CNN2 operates with 32 input channel. After a careful analysis of the CNNs outputs for the entire dataset, all thresholds can be quantized to an 8-bit dynamic fixed point as this bit-width offers enough range to represent the values obtained from (7.11). Hence, the binarizer is reduced to a filter-specific threshold comparator whose thresholds are derived from model parameters, effectively reducing the hardware implementation complexity.

Figure 7.3: Binarizer architecture

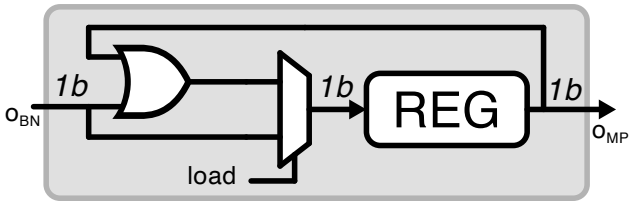


Source: (ROCHA et al., 2020)

The binarizer unit features a register file to store the thresholds and a comparator, as illustrated in Figure 7.3. Both input data and thresholds are m-bit wide, and each binarizer unit can have p comparators in parallel. Further, p determines the number of input channels and RF read ports connected to each comparator. According to (7.11), each CNN filter will have an associated threshold which is accessed using the same filter address used on the CNN module. When $p \geq 1$, only the first filter address is informed as the RF controller assumes that the remaining addresses are contiguous to the first one, so the thresholds are outputted in a burst-like reading fashion. Support to parallel comparison is mandatory to cope with the required throughput after at different pipeline stages. While CNN1 outputs one value per clock cycle, CNN2 outputs four values per cycle, requiring four parallel comparators to avoid pipeline stalling and extra temporary memory.

All data is binarized before the max-pooling layer as it reduces the implementation complexity of the former. For a non-binarized approach, the pooling unit requires a comparator, like the one employed on the binarization, regardless of the pooling strategy (minimum, average, or maximum). Conversely, the pooling unit implementation can be simplified to a single logic gate if the inputs are binarized. Hence, as the CorNET model employs max-pooling layers, the hardware implementation of these modules is shown in Figure 7.4, and it features a multiplexer, an OR gate, and a register.

Figure 7.4: Binary max-pooling unit



Source: (ROCHA et al., 2020)

This architecture employs a serial approach to compute the 4:1 max-pooling. A new value is directly written to the register every four cycles, and it represents the first value of the 4:1 pooling kernel, effectively bypassing the OR gate. In the next three cycles, the multiplexer selects the comparator value computed from the current input and current register value to update the register. The feedback loop, along with the OR gate, ensures that if one of the four values is one, the max-pooling output will be one.

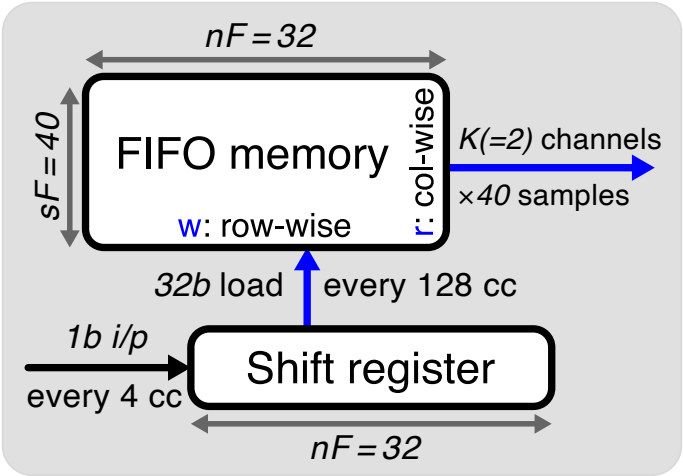
7.2.3 Transposition Buffer

Although the CorNET model employs 1-D PPG windows of 1000 samples, the CNN1 output feature map has 32 channels, one per filter, and each channel contains 961 convolved values. Binarization and pooling effectively reduce the number of samples in each channel to 240 ($\lfloor {}^{961}/_{4} \rfloor$) values so that the feature map can be seen as a matrix of 32 columns and 240 lines. As the hardware implementation of the first CNN adopts an input reuse scheme, this feature map is generated in a sample-wise fashion due to the max-pooling, i.e., all the values in the same row must be computed before moving to the next row. Nonetheless, the CNN2 layer requires that each input channel has at least 40 elements corresponding to the filter size (sF). In this case, each filter computation must read the values from all input channels in a channel-wise manner.

This problem is addressed using a transposition buffer, illustrated in Figure 7.5,

which features a row-wise writing scheme by the CNN1 with a column-wise reading mechanism by the convolutional filters of CNN2. The buffer maximizes the data reuse and improves the system latency as it enables the CNN2 computation before the entire CNN1 output feature map is completely computed.

Figure 7.5: Transposition buffer architecture



Source: (ROCHA et al., 2020)

The transposition buffer features a 32-bit shift register and a FIFO-based memory of size 32×40 bits able to hold 40 values for each one of the 32 CNN1 filters. The input interface has a 1-bit data signal connected to the shift register that receives a new data value every four clock cycles from the 4:1 max-pooling unit, which corresponds to each CNN1 filter output. After $4 \times 32 = 128$ cycles, the register holds a new valid value, and the buffer controller writes this value to the memory. Further FIFO writes will occur every time that the current value of the shift register has been completely overwritten. Once the FIFO is full, the controller sends a signal to the CNN2 to initiate the data processing. This process is repeated until the current PPG window has been completely processed, i.e., the 240 samples of each input channel have been written to the transposition buffer.

Since the FIFO memory is updated every 128 cycles, it constraints the CNN2 block execution time to 128 cycles to process the entire memory contents, hence the output interface has K parallel channels (columns) where each is 40 bits wide (corresponding to the filter size). For each data read request, the controller selects K columns from the FIFO and increments the internal read counter to keep track of how many valid columns are left. Although K can assume any value between 1 and 32, this implementation assumes $K = 2$ as it ensures the memory throughput required by CNN2 while minimizing the interface requirements.

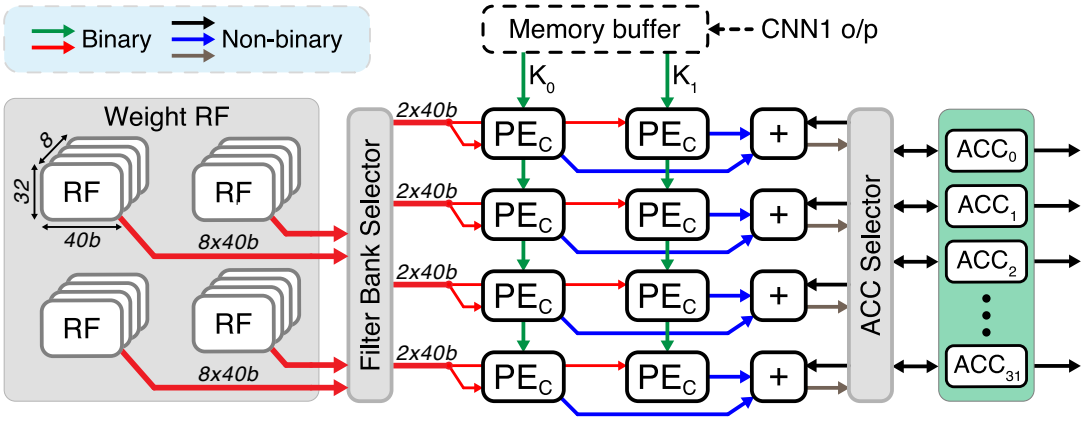
7.2.4 CNN2 Layer

Increasing the number of input channels from 1 to 32 poses an enormous computing challenge, even with the reduction of data dimensionality due to max-pooling. Further, the stream-based architecture dictates that the CNN2 must process the entire input data batch will all input channels in 128 cycles before the memory buffer is refreshed, and the computation moves to the next step. Each data batch read from the memory buffer corresponds to an output line on the CNN2 output feature map. The computation cost associated with this operation can be computed according to (7.12) where C_{in} is the number of input channels (in this case, 32), sF is the filter size (40 samples), and nF is the number of filters in this layer, which is also 32.

$$\#\text{OP} = C_{in} \times sF \times nF = 32 \times 40 \times 32 = 40960 \tag{7.12}$$

Completing these operations within the execution time constraint requires the module to be capable of computing $40960/128 = 320$ operations per clock cycle. Since each filter has 40 coefficients and 32 filters/input channels on this layer, this throughput can be achieved with intra- and inter-filter parallelization. If all 40 coefficients are computed in parallel, the 320 operations/cycle is achieved with eight parallel filters. Figure 7.6 illustrates the proposed architecture composed of eight ($M = 8$) convolution processing elements (PE_C) in parallel, a register file for weight storage, a set of adder trees and an accumulator bank.

Figure 7.6: Internal CNN2 hardware architecture



Source: (ROCHA et al., 2020)

This architecture adopts a hybrid data access scheme, which enables the processing of multiple convolution filters (F_*) over multiple input channels (K_*). Choosing the

right number of parallel filters and input channels is quintessential for optimal resource allocation since there are eight PEs available. Further, F and K determine the communication bandwidth for the register file and memory buffer, respectively. These parameters are defined in terms of M as $F = {}^{M}/_{I}$ for $I \in [1, 2, 4, 8]$. Table 7.1 shows an analysis of multiple combinations of F and K considering bandwidth requirements, the latency per output, and memory accesses and data reuse metrics. This analysis shows that the hybrid scheme offers the optimal solution as it maximizes both weight and input data reuse; hence, the $(4, 2)$ pair was chosen as it reduces the external communication requirements with the memory buffer. The PE_C on the same line belongs to the same filter (red line), and they are applied on different input channels (green line) whose result is accumulated with the previous values of the respective accumulator (selected from the green box). Each input channel is shared among all processing elements on the same column.

Table 7.1: Resource allocation analysis on CNN2 architecture

	(F, K) combination		
Metrics	**Input stationary (8, 1)**	**Weight stationary (1, 8)**	**Hybrid (2, 4) & (4, 2)**
Input interface width (bits)	1×40	8×40	$I \times 40$
1st output latency (cycles)	$31 \times 4 + 1 = 125$	5	$({}^{32}/_{I} - 1) \times \frac{32 \times I}{M} + 1$
# Input accesses	$32 \times 4 = 128$	$32 \times {}^{32}/_{8} \times 4 = 512$	$\frac{32 \times 4}{I}$
# Weight accesses	$32 \times {}^{32}/_{8} \times 4 = 512$	$32 \times 4 = 128$	128×4
# Input reuse	32	1	${}^{M}/_{I}$
# Weight reuse	1	32	I
Memory access schedule	Every 4 cycles	Every cycle	Every $\frac{32 \times I}{M}$ cycles
Weight access schedule	Every cycle	Every 4 cycles	Every cycle

Source: The Author

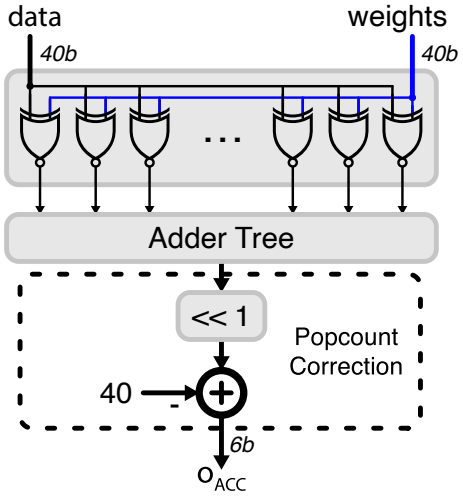
As both inputs and weights are binarized (bipolar variables), the multiplication can be implemented with XNOR operations along with a bit counting hardware (KIM; SMARAGDIS, 2016). The XNOR gate is a simpler and faster substitute for the binary multiplication due to its truth table, and computing the number of ones on the XNOR output gives the multiplication result. Yet, mapping the $[-1, +1]$ pair to $[0, 1]$ causes a shift on final result which must be compensated with Equation 7.13 where *popcount* is the XNOR result bit count and NB is the input bit-width.

$$A_b \times B_b = 2 \times \text{popcount}(XNOR(A, B)) - NB \tag{7.13}$$

Each PE_C module implements a multiplication of 40-dimensional binary vectors

using the popcount algorithm. Figure 7.7 depicts the PE_C architecture, and it is divided into three blocks: (i) XNOR array: performs binary multiplications; (ii) a carry save-based adder tree to accumulate the multiplication results (popcount), and (iii) fixed shift-add block to compensate the bias added by popcount.

Figure 7.7: The internal architecture of a PE_C



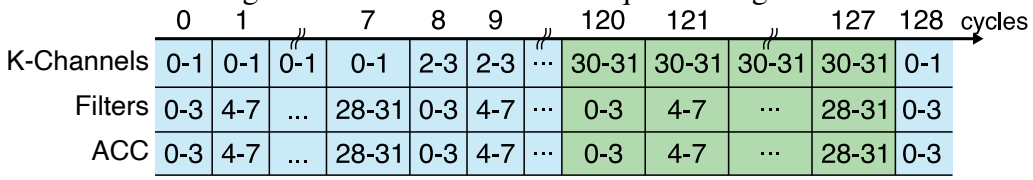
Source: (ROCHA et al., 2020)

Since each convolution filter has different weight values for each input channel, the weight register file stores 40960 bits. The RF contains 32 banks (divided into four segments, each having eight groups) where each bank has 32 lines of 40 bits. As the architecture computes four filters over two input channels, the RF controller takes two addresses to select the eight PE_C weight sets. The first address selects the group within a filter segment while the second address selects the line from that particular bank, which corresponds to the input channel being processed. Each weight data request occurs in a burst-like manner, i.e., the controller outputs the four contiguous filters – filters 0 to 3, for instance – for the two adjacent input channels – channels 2 and 3.

The architecture favors the input reuse scheme, so when the memory buffer sends the ready signal, the CNN2 controller loads the weight set corresponding to filters 0-3 and input channels 0-1 into the PE_C, as shown in Figure 7.6. The filter computations are accumulated with the respective accumulation register, which is reset every time the memory buffer is updated. Once the computation of these filters is done, the controller rolls over and loads the weight set corresponding to filter 4-7, considering the same two input channels (0-1), accumulating the results on the appropriate registers. Figure 7.8 illustrates the execution flow for all filters and input channels. Given that four filters are computed per clock cycle, and there are 32 filters, the current input channels are processed in $^{32}/_{4} = 8$ cycles. Once the current input channels have been processed, the

controller requires new data from the buffer and starts the process again without resetting the accumulators. Since each input data request fetches two input channels per clock cycle, this process is repeated $^{32}/_{2} = 16$ times.

Figure 7.8: CNN2 execution sequence diagram



Source: (ROCHA et al., 2020)

The last two feature maps are loaded 120 clock cycles after the CNN2 computation start. At this point, all the filters have already accumulated the multiplications of previous channels, and the filters 0-3 are the first ones to be processed, which means that their final value will be available at the clock cycle 121. The remaining filters will be processed consecutively, and the accumulators 28-31 will be updated with the final value on the clock cycle 128, as shown in the green block in Figure 7.8. As soon as the first accumulators have their updated result (cycle 121), the binarization process can start before the second max-pooling.

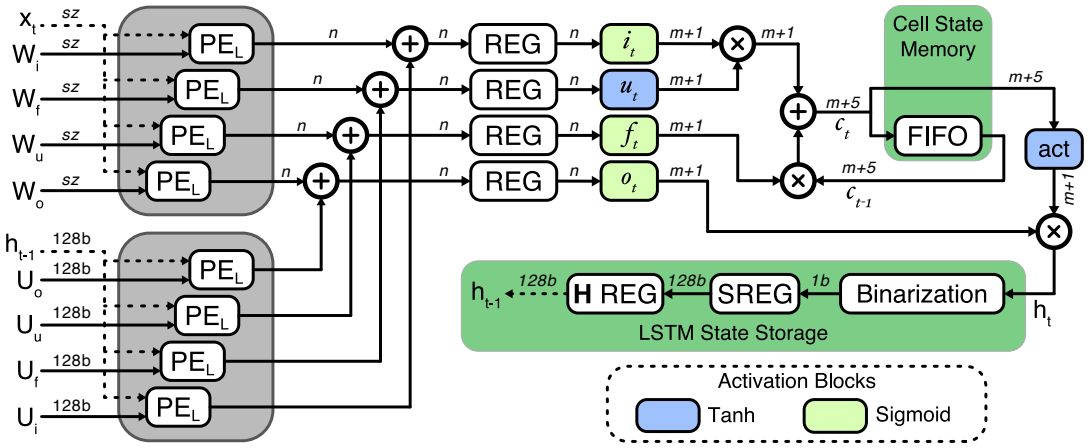
The interface between the CNN2 block and the binarizer outputs four accumulators at the time due to the number of filters computed in parallel, which entails in a binarizer block with four comparators in parallel to cope with the system throughput. Further, there are 32 1-bit max-pooling units to store the intermediary pooling values of each output filter as the CNN2 architecture does not generate four outputs in-a-row like the CNN1 module. In this case, each max-pooling unit will receive a new value every 128 cycles, which is the time frame to process one output row, so the units must keep the values between updates. Adopting parallel units is an area and latency trade-off since adopting the same CNN1 sequential outputs per filter would require adding 96 bits to the memory buffer (7.5% area increase), which would result in a latency penalty of 384 cycles. The final result on the 4:1 pooling blocks, i.e., the current timestep vector for the LSTM layer is obtained after processing four CNN2 outputs ($4 \times 128 = 512$ cycles). Note that the max-pooling registers are used as input buffers for the bLSTM1 layer.

7.2.5 Binary LSTM layer

Long short-term memory layers have increased implementation complexity due to their recurrent nature, which requires intermediary data storage. These layers operate over two input data sources: the timestep input (x_t) – generated by the CNN2 output and stored on the max-pooling registers – and the previous hidden state (h_{t-1}) – stored internally to the layer – to compute the current cell and hidden states. On the bCorNET framework, both bLSTMs share the same number of neurons, although the first layer assumes an input timestep width (sz) of 32 bits while the second one assumes that sz is equal to 128.

Since both binary LSTM layers adopt the same binarization scheme and number of neurons, they share the same hardware architecture, shown in Figure 7.9. According to the binarization equations in Section 7.1.2, each layer is divided into three serially attached computation modules: binary MAC computation, gate non-linearity operations, and state/output computations. Further, it has eight register files for weight storage and eight registers for weight-scaling storage, which are not shown in Figure 7.9. The register file in the first LSTM is composed of four $32b \times 128$ blocks and four $128b \times 128$ blocks. Similarly, LSTM2 has eight similar blocks of 128×128 bits.

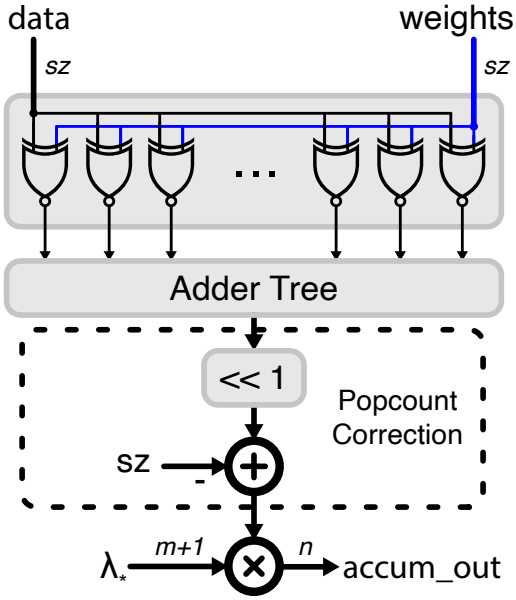
Figure 7.9: Binary LSTM architecture for both LSTM layers in bCorNET



Source: (ROCHA et al., 2020)

This architecture adopts a mixed data representation scheme because even though the input and weight binarization, the gates' outputs cannot be binarized without severely impacting the network accuracy. The LSTM layer employs a fixed-point implementation whose precision is fine-tuned to obtain an optimal trade-off between quantization-induced accuracy loss and circuit performance. Hence, both input and output interfaces are binary, while the internal computations are represented in a Qn.m fixed-point format.

Figure 7.10: Internal architecture of a PE_L



Source: (ROCHA et al., 2020)

The binary MAC computation section has eight binary processing elements (PE_L) – shown in Figure 7.10) – along with four adders to combine the convolved inputs with the respective convolved hidden state. Each PE_L multiplies the binary input (either x_t or h_{t-1}) by the associated binary weights (W_* or U_*). As both layers have 128 neurons (or LSTM units), there are 128 different weight sets for each (x_t, h_{t-1}) pair, and each pair is processed in one cycle. Note that the PE_L architecture is similar to PE_C (see Figure 7.7), although it features a fourth block where the output is multiplied by the scaling factor (λ_*), ignored on the CNN modules.

As each MAC computation is finished, the results are combined using a carry-propagating adder before the pipeline register. The weight-scaling parameters define the adder width as they scale the popcount output in each PE_L. In this segment, n is set to 25 bits to accommodate the integer part (8 bits) and the fractional part (17 bits). The former is defined according to the popcount maximum value while the latter is derived from the non-linear computation blocks. All adders are based on a carry select architecture with variable group size as it offers the best trade-off between speed and area.

While CNN activation functions could be binarized, all non-linear (NL) operations on the LSTM gates are based on sigmoid and hyperbolic tangent functions, which are expensive to compute and cannot benefit from binarization techniques (TIMMONS; RICE, 2020). Hence, these functions were implemented using lookup tables (LUT) aiming for a better trade-off between circuit area footprint and approximation error. The accuracy of these non-linearities are highly affected by the bit-width, so they define most of the

datapath bit-width. Since the internal datapath adopts the Qn.m fixed-point representation, the optimal values for n and m are determined based on heuristic analysis of three parameters relative to the LUT design: (i) number of LUT segments; (ii) fraction size (m) and (iii) maximum input data range ($n - m$). Further, the sigmoid function can be computed as a function of the hyperbolic tangent, according to (7.14), simplifying the LUT implementation.

$$\sigma(x) = \tanh\left(\frac{x}{2}\right) + 0.5 \tag{7.14}$$

The quantization step size dictates the minimal distance between two input values (x value) to determine the output value, and it is determined by the inverse of the number of segments. The number of segments is constrained to a power of 2, constraining the step size to a negative power of 2 that can be implemented as a shift, leading to simpler hardware. In this exploration, LUTs with 32, 64, and 128 segments were tested. Further, the input range dictates how small the step size as the latter is a given by (7.15). The input range lower bound is set to 4 as lower values would affect the accuracy near the function edges considerably, and the upper bound was limited to 8 as the accuracy improvements would require a non-optimal number of fraction bits.

$$\text{Step size} = \frac{\text{Maximum input range}}{\text{Number of segments}} \tag{7.15}$$

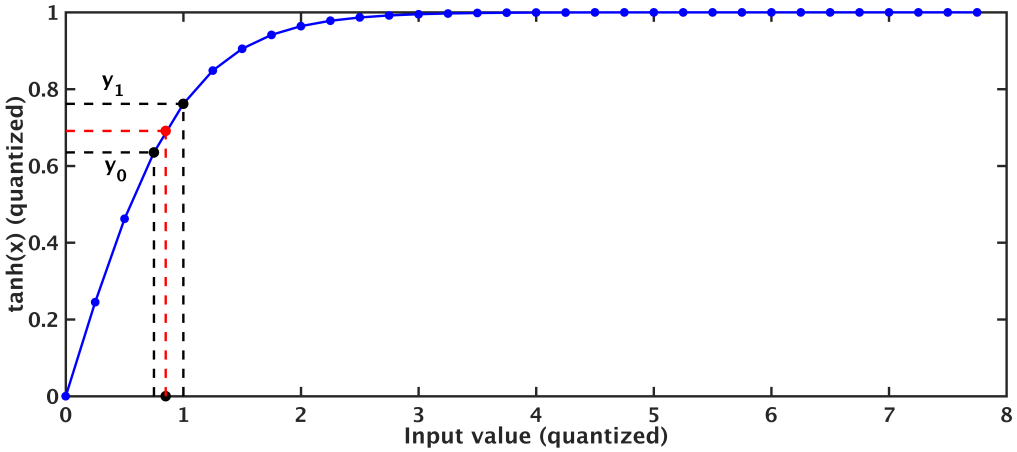
The step size alone cannot determine the LUT accuracy as it only dictates the input quantization. Since the outputs of both sigmoid and hyperbolic tangent functions are constrained to $[0, 1]$ and $[-1, 1]$, respectively, the fraction size (m) will determine the output quantization step size. The actual word width on each LUT block is set to $m + 1$ to account for the sign bit as the values are represented in two's complement. This exploration considered values of m ranging from 6 to 18 bits.

For improved accuracy, the LUT employs a linear output interpolation, which does not require any complex arithmetic operator since the division process is implemented using an adder and a shift as the step size of the x-axis is a negative power of two. The interpolation is implemented according to Equation 7.16 where y_0 is the LUT segment corresponding to the input (i_{LUT}), y_1 is the next segment on the LUT, and x_0 is the largest quantized input smaller than i_{LUT}. Note that the difference between x_0 and x_1 is always

a negative power of 2, so the division is reduced to a shift operation.

$$\overline{\tanh_{LUT}}(i_{LUT}) = y_0 + (y_1 - y_0)\left(\frac{i_{LUT} - x_0}{x_1 - x_0}\right) \tag{7.16}$$

Figure 7.11: LUT interpolation example for the tanh function



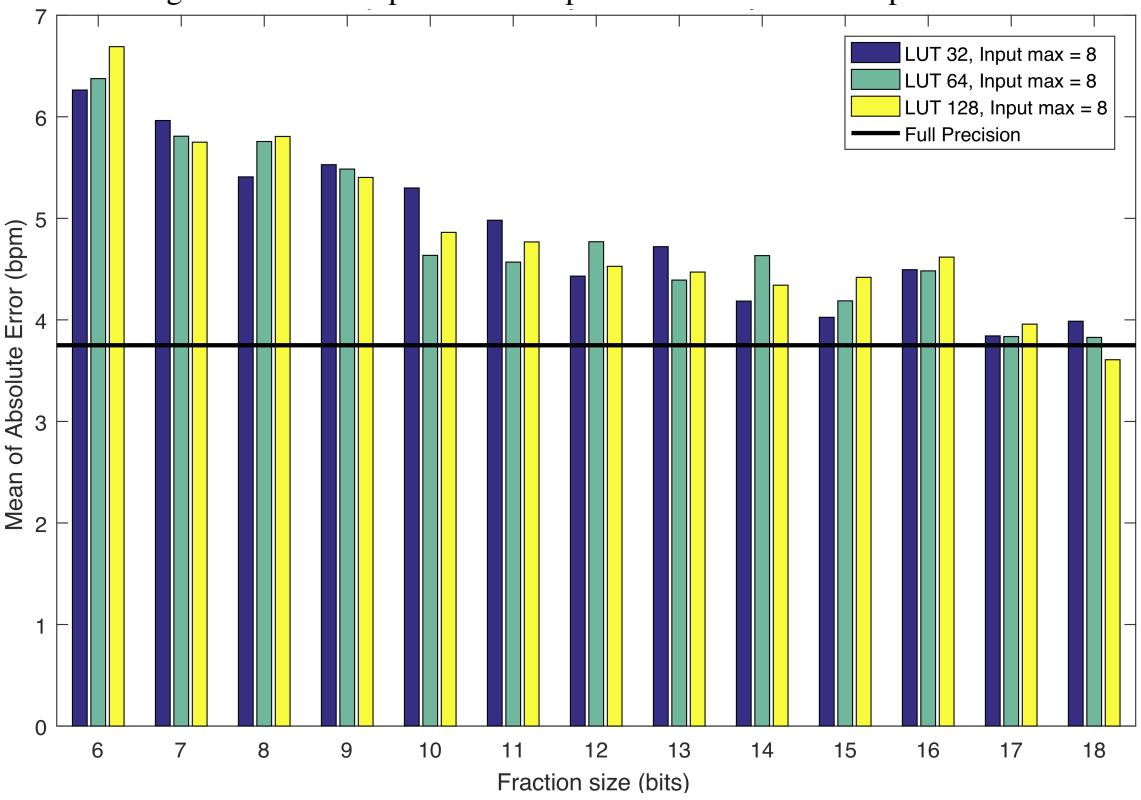
Source: The author

Figure 7.11 illustrates how the interpolation is computed assuming $i_{LUT} = 0.85$. This input value falls between the 0.75 and 1.00 quantized points that have associated output values (y_0 and y_1, respectively) on the LUT. At this point, the accuracy drop is reduced from 8.1% to 0.77% when comparing the baseline LUT implementation with the interpolation-based implementation.

Nevertheless, the non-linear function accuracy itself is not the best metric to determine the best set of parameters for the LUT implementation. Then, Figure 7.12 shows the mean absolute error of the predicted HR over the first 12 data recordings on the training dataset for the exploration of a different number of segments, fraction size, and input range. The black line represents the full-precision software implementation of the activation functions. The optimal accuracy-area trade-off is given by a LUT implementation with 32 segments, $m = 17$ bits for the fraction size and 5 bits for the integer part on the input representation, i.e., the value of n is $17 + 5 = 22$ bits.

The hyperbolic tangent LUT implementation follows the architecture shown in Figure 7.13. It features a ROM to store the quantized tanh values, an input/output limiter block (in blue), and the output interpolation circuit (in gray). Due to the hyperbolic tangent symmetry, the ROM memory only needs to store half the quantized curve points. Ensuring the entire output range requires computing the absolute value of the input and determine the input sign (ABS). Then, the absolute value is used to access the ROM memory, while the sign is used to correct the output sign through a two's complement

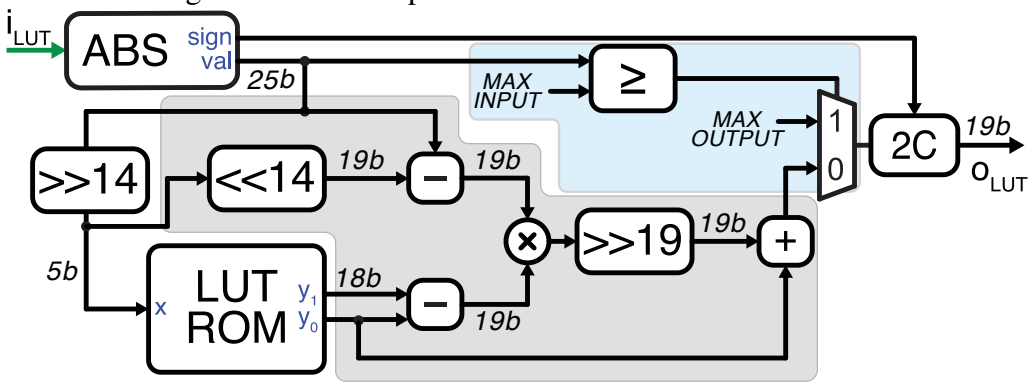
Figure 7.12: Fixed-point LUT implementation trade-off exploration



Source: (ROCHA et al., 2020)

computation block (2C).

Figure 7.13: Lookup table architecture for tanh function



Source: The author

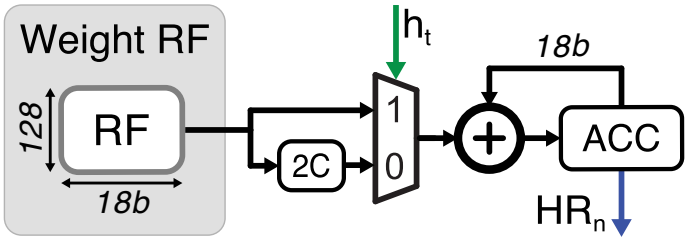
The light blue block on Figure 7.13 checks whether or not the input (i_{LUT}) is outside the range used to compute the values stored on the ROM memory. If the input magnitude is larger than the maximum range of the tanh x-axis (8, in this case), the circuit in gray is bypassed, and the output magnitude is rounded to 1 (in fixed-point format), then the output sign is corrected accordingly. If the i_{LUT} value is within the valid range, the five most significant bits are used to select the segments y_0 and y_1, where y_0 is the quantized tanh point corresponding to the current i_{LUT} value and y_1 is the next point on the ROM

memory. The gray block implements the output interpolation circuit, according to (7.16).

7.2.6 Quantized dense layer

The regression layer is responsible for translating the computed hidden state of the bLSTM2 into a real-valued HR. Opposite to other layers in the system, this single neuron dense layer relies on a MAC operation with binary input data and quantized weight values. The weight quantization follows the bit-width constraint determined by the LUT implementation. For optimal resource allocation, the dense layer architecture, illustrated in Figure 7.14, is based on a serial approach to match the data production rate of bLSTM2. It features a register file to store 128 weights of 18 bits, a multiplexer to select between the stored value and its 2's complement value and an accumulator. This accumulator will not overflow since the output is normalized following the training methodology.

Figure 7.14: Quantized dense layer architecture



Source: The author

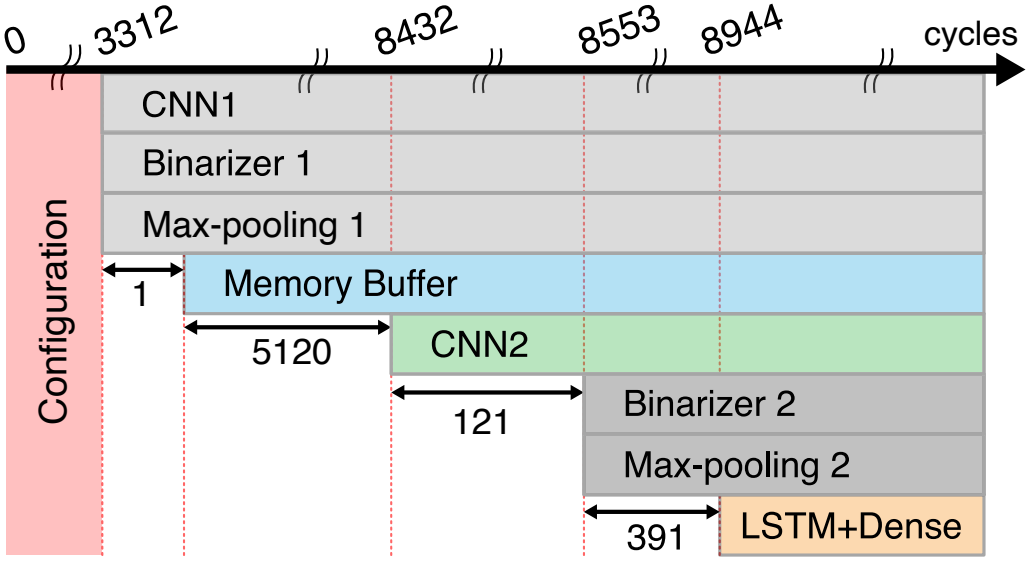
As each value of the last bLSTM2 hidden state (h_t) is computed, this value is also fed to the quantized dense layer to determine the multiplication sign. Since a new input value is available every cycle, an internal counter generates the address to access the weight memory. After 128 cycles, the accumulator has the updated normalized heart rate (HR_n) for that input PPG window. The accumulator adder employs a carry-select adder for optimal balance between performance, area, and power consumption.

7.3 Timing Analysis

The primary target of the bCorNET framework is on-node processing, where the overall system performance can be assessed in terms of latency and throughput. The system operation is divided into configuration and evaluation phases, as illustrated in Figure 7.15. During the configuration phase, all the weights, thresholds, and scaling factors are

serially loaded into the respective register files, and the system top level controller manages the entire process. Sequential loading of the network parameters helps saving memory bandwidth and reduces the interface requirements with a negligible latency penalty as it happens only once.

Figure 7.15: Overall architecture timing sequence



Source: (ROCHA et al., 2020)

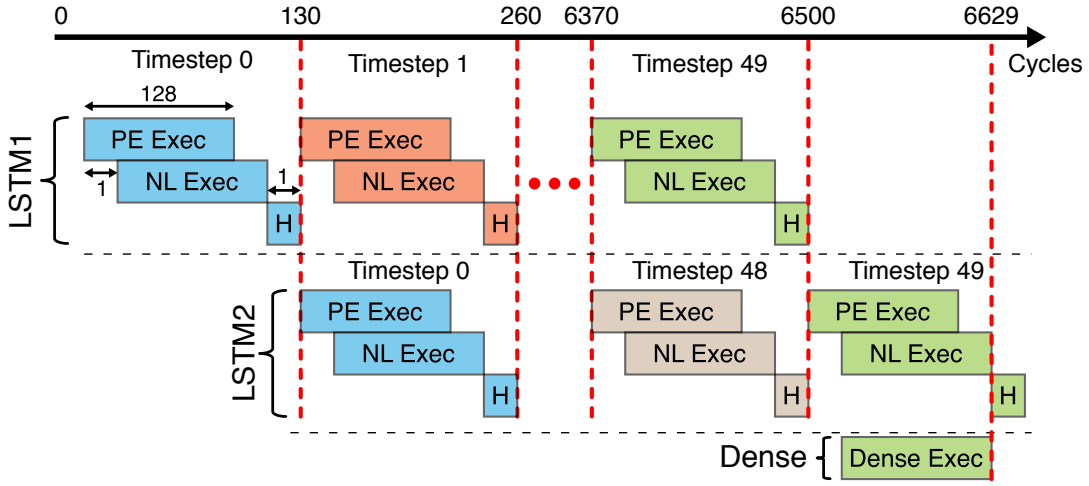
Once all the modules have been properly configured, the system changes to the evaluation phase, which will set the system to predict an HR based on the current PPG window. Based on the stream architecture concept, the system adopts a 6-stage pipelined design – CNN1 bundle, transposition buffer, CNN2, max-pooling, LSTM1, LSTM2 – to achieve a balance between speed and system latency. The CNN1, binarizer, and sequential max-pooling modules comprise the first pipelining stage, so the computation happens on the same cycle, which helps saving latency cycles at the expense of a slightly longer critical path. Each filter has a valid max-pooled output every four cycles when it is ready to be stored on the transposition buffer. Since the buffer holds 32×40 bits, it takes 5120 (i.e. $4 \times 32 \times 40$) cycles to be filled and ready to output data.

After the buffer is filled, the CNN2 module is enabled, and it has 128 cycles to process the entire buffer content to obtain one valid output for each filter. Differently than CNN2, the second convolutional layer and max-pooling units are two separate pipeline stages. Hence, a valid LSTM1 input timestep is ready on the register of the second max-pooling layer after 4 CNN2 outputs have been processed, which is equivalent to $4 \times 128 + 1 = 513$ cycles. Considering the buffer filling (5120 cycles) and the CNN2 with max-pooling (513 cycles) latencies, the first LSTM layer can start processing the first timestep

after $5120 + 513 = 5633$ cycles from the start of the evaluation phase. At this point, the first four stages are full of data.

LSTMs are considerably more complex due to their intra-layer pipelining and the recurrent nature. Each bLSTM layer sequentially processes its 128 neurons according to the timing diagram shown in Figure 7.16, assuming that all the timesteps are ready to be processed. It is important to note that despite the serial processing strategy adopted on the LSTM layers, they finish the timestep processing before the CNN2 max-pooling has available data on the output, requiring a pipelining stall between timesteps, achieved with clock-gating for power saving. Although stalling the pipeline is not optimal, it is a trade-off between the computation capability of the LSTM layers and the additional memory for a more serialized architecture.

Figure 7.16: Timing diagram for bLSTM layers



Source: (ROCHA et al., 2020)

Each LSTM layer has a two-stage pipeline for MAC computation (PE exec) and non-linear processing (NL exec). All PE_L are activated once the data is available on the input, processing one neuron per clock cycle. After 128 cycles, the hidden state shift register (SREG) has the updated hidden state, and its contents are copied to HREG to be used on the next timestep.Due to their recurrent nature, both LSTM layers must complete processing timestep t before it starts processing the $t + 1$ timestep since the hidden state h_t is required to compute h_{t+1}. Therefore, each layer completes a timestep processing in 130 cycles. Once the HREG is updated on bLSTM1, the bLSTM2 is activated, effectively all system pipeline stages. The dense layer is activated when the bLSTM2 start the NL processing of the last timestep to reduce latency as it also adopts a serial processing strategy.

The total system latency (SL) is given by the complete processing of an 8s PPG

window, i.e., the system has to fill the memory buffer, process the 240 lines of the max-pooled CNN1 output feature map and one timestep by each bLSTM layer. Note that the last CNN2 input feature map line is not considered due to the LSTM pipelining. Hence:

$$SL = 5120 + (240 - 1) \times 128 + 260 = 35972 \text{ cycles} \tag{7.17}$$

7.4 Evaluation of the bCorNET framework on ASIC and FPGA platforms

The bCorNET framework evaluation has two key aspects: (a) accuracy measurements on the binarized/quantized model with LUT-based non-linear functions, and (b) performance comparison on both FPGA and ASIC platforms. For the accuracy measurements, the bCorNET framework was trained and evaluated using the methodology presented in Section 6.2.2. For the hardware evaluation, each platform has a specific methodology to be followed.

The evaluation on 22 subjects, comparing bCorNET using both software and hardware models, is shown in Table 7.2. Despite the quantization of both non-linear functions and weight scaling factors, the accuracy drop is about 9.7%, moving from an MAE of 6.67±5.49 to 7.32±5.68. Since the LSTMs operate on a recurrent fashion, considering the value from the previous iteration, the quantization error is accumulated through

Considering the RTL-based evaluation models, subject 23 shows the best results (2.92±2.13) whose HR estimation is shown in Figure 7.17. Note that the quantization error can improve the accuracy as illustrated by subject 23, which has a 7.2% lower MAE if compared with the full precision software version.

Conversely, the worst result is presented by subject 8 (15.00±11.26), whose HR estimation is shown in Figure 7.18. The intense motion artifact on this subject poses a severe challenge for the model to differentiate noise from the actual HR component on the PPG data. However, the first 12 records showed the lowest error on average as they have more subjects executing the same activity.

7.4.1 Hardware Synthesis

All bCorNET hardware modules were described in Verilog, and the same design files were used for both target platforms. For a proof-of-concept demonstration for real-

Table 7.2: Binary CorNET Evaluation

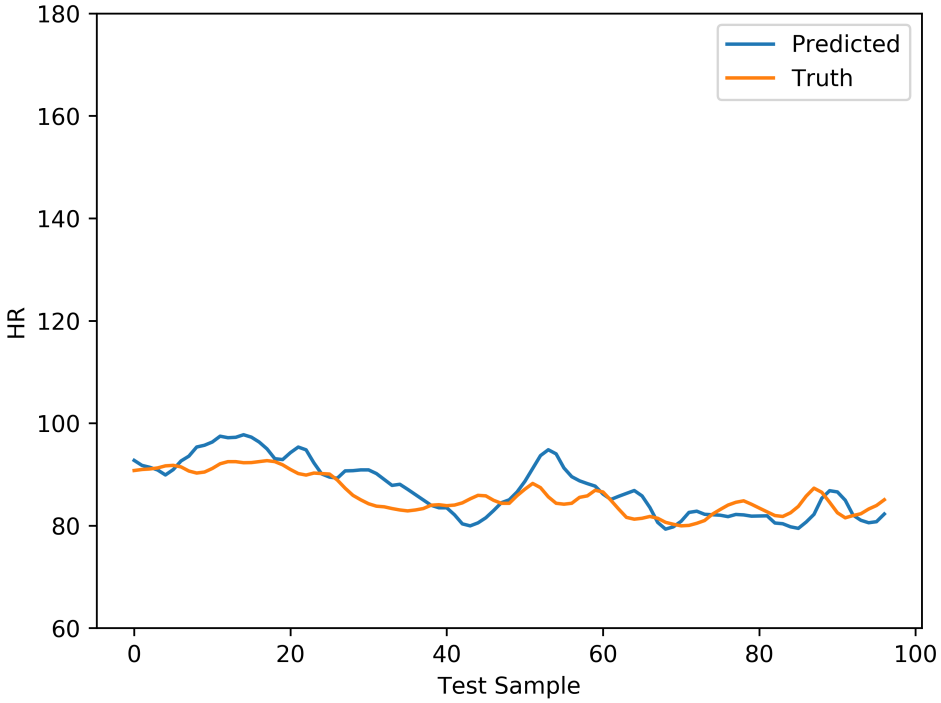
Record	Binary CorNET	Binary CorNET (RTL)
1	5.70 ± 4.14	5.80 ± 4.07
2	9.51 ± 7.89	9.38 ± 9.19
3	5.52 ± 4.29	5.37 ± 3.24
4	3.75 ± 2.96	4.48 ± 3.95
5	3.18 ± 2.58	4.44 ± 2.70
6	3.38 ± 3.30	4.20 ± 3.47
7	2.49 ± 1.67	3.53 ± 2.76
8	13.30 ± 11.77	15.00 ± 11.26
9	3.17 ± 2.30	3.86 ± 2.90
10	12.84 ± 9.24	13.12 ± 11.74
11	7.42 ± 5.46	7.92 ± 4.74
12	4.17 ± 3.75	4.22 ± 3.49
13	-	-
14	11.19 ± 9.17	12.90 ± 12.22
15	5.39 ± 4.67	6.66 ± 5.66
16	8.56 ± 7.15	8.77 ± 7.89
17	13.63 ± 11.53	13.22 ± 11.40
18	5.08 ± 3.52	5.92 ± 4.51
19	3.16 ± 3.09	3.54 ± 3.37
20	6.09 ± 5.45	8.15 ± 7.15
21	10.25 ± 9.49	11.83 ± 9.83
22	5.76 ± 5.24	5.79 ± 4.37
23	3.15 ± 2.12	2.92 ± 2.13
Record 1-12 (T1)		
MAE±SDAE	**6.20 ± 4.95**	**6.78 ± 5.29**
Record 13-23 (T2 and T3)		
MAE±SDAE	**7.23 ± 6.14**	**7.97 ± 5.97**
Record 1-23 (T1, T2 and T3)		
MAE±SDAE	**6.67 ± 5.49**	**7.32 ± 5.68**

Source: The Author

time operability, the design was synthesized for FPGA using the Xilinx Vivado Design Suite v2018.3 synthesis tool for a Kintex 7 (XC7K70T) device. For this platform, the design explores device-specific features like DSP modules and block RAM (BRAM) units to improve the synthesis quality of results with a smaller area and better timing. Table 7.3 reports the bCorNET accelerator synthesis results, showing it is able to achieve real-time HR computation with a small circuit footprint. There are no works on the literature that

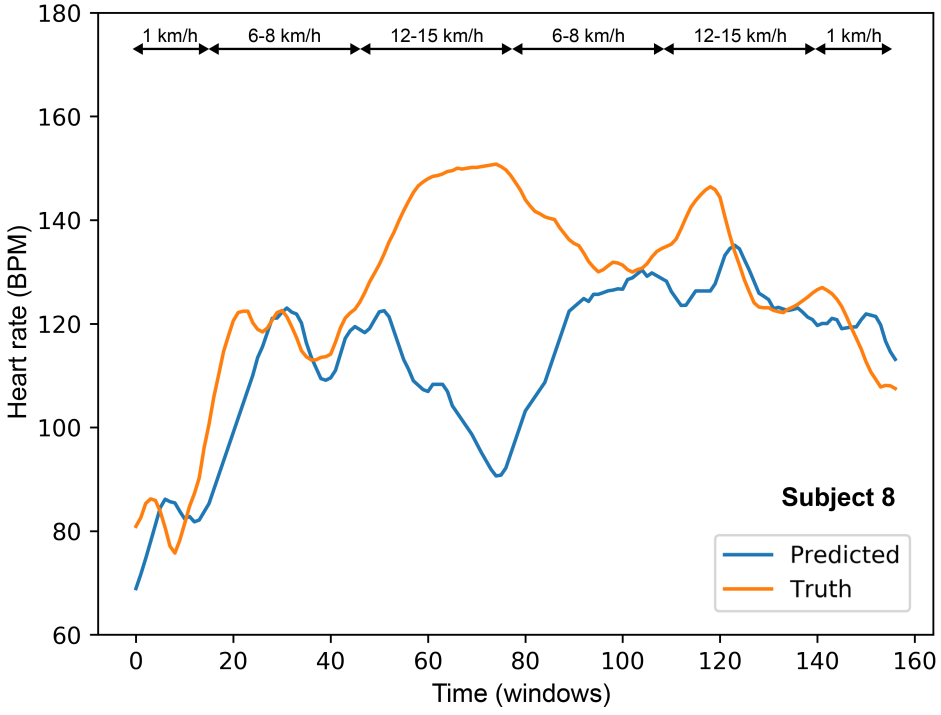
propose a joint CNN-LSTM accelerator for a fair comparison. Yet, Table 7.3 also reports the circuit usage for other similar architectures to clarify the clear benefits of quantization

Figure 7.17: PPG Estimated vs true ECG HR for Subject 23



Source: The author

Figure 7.18: PPG Estimated vs true ECG HR for Subject 8



Source: The author

and binarization on the device resource usage with respect to non-binary implementations.

Table 7.3: FPGA Implementation results

	(CHANG; CULURCIELLO, 2017)	(GUO et al., 2017)	bCorNET
Input Precision	16b Fixed	12b Fixed	5b Fixed
Network Arch	LSTM	Variable LSTM	bCNN+bLSTM+qDense
Platform	XC7Z045	XCKU060	XC7K70T
LUTs	61834	294000	100250
DSPs	-	1505	54
BRAMs	-	119	21
GOPS	-	2520	27
Frequency (MHz)	142.0	200.0	10

Source: The Author

Conversely, the ASIC implementation flow does not rely on any technology-specific memory cores to optimize register files and FIFO memories. Therefore, all memory-related components are mapped to registers available on the standard cell library. The synthesis flow was performed using the Cadence Genus™ synthesis tool (CADENCE, 2018) considering a standard cell library from an ST 65nm process with standard V_t transistors operating with a supply voltage of 1.0V. For accurate power estimation, the methodology proposed in (PAIM et al., 2019b) was adopted with PLE mode activated on the synthesis tool.

The target frequency was set 1 MHz for two reasons: (a) several biomedical sensor platforms have internal components operating at this speed, like in (KONIJNENBURG et al., 2016; SCHONLE et al., 2017), and (b) considering the total system latency, it still offers a real-time capability for HR inference as the computation would be finish in about 36ms for each PPG window. Table 7.4 summarizes the synthesis results for the bCorNET framework. There are no works on the literature that proposed a similar hardware accelerator for an ASIC platform prohibiting any comparison.

7.5 Chapter Summary

This Chapter described bCorNET, a model that embedded modifications on the original CorNET model to make it suitable for running on edge devices. This is achieved by the combination of quantization and binarization techniques to reduce the memory and computational requirements.

Further, it presented a tailored stream-based hardware architecture to maximize the efficiency of the bCorNET model in terms of resource requirements, energy consumption

Table 7.4: ASIC Implementation results

Data precision	5-bit (inputs) / binary (weights)
Network Arch	CNN+LSTM+Dense
Process	65 nm
Energy/window (μJ)	56.1
Mem. (kbits)	260
Cell Area (μm^2)	3399657
Gates (NAND2 Eq.)	1634450
Op. Frequency (MHz)	1

Source: The Author

and throughput. The circuit has a small area footprint and it requires a very low clock frequency to achieve real-time operation. The design was synthesized for FPGA platform as a proof-of-concept, and for an ASIC platform to assess the circuit characteristics for embedding it into a wearable device.

8 CONCLUSIONS

The work developed in this thesis led to contributions to the field of arithmetic operators as well as in the design of efficient hardware architectures for embedded neural networks. This chapter concludes this thesis, pointing the main contributions, and discussing future directions for this research.

8.1 Main findings

Chapter 5 presented a highly flexible framework to generate RTL designs for arithmetic operators. The modular architecture simplifies the framework extensibility needed to support new algorithms that may be used in arithmetic operations. Due to the algorithmic-level description of circuits and the automatic testbench generation system, the framework proves to be a very efficient tool for digital designers to explore design alternatives to fulfill specific design requirements. The framework efficiency was assessed through the generation of several multiplier combinations that were synthesized considering a commercial technology and synthesis flow. Results showed that state-of-the-art EDA tools do not have optimized multiplier-aware mapping algorithms. Further, its flexibility allowed the optimization of the Radix-2^m multiplier, resulting in a smaller, more energy-efficient architecture than the baseline implementation. The framework is a relevant contribution to the community as it publicly available for use by digital designers looking for optimized arithmetic circuits.

Chapter 6 presented an alternative training methodology for the CorNET framework, allowing a more generalized approach that is more suited for deployment in embedded devices. Further, considering other subjects on the training dataset makes the model more prone to capture the subtle physiological differences in each subject.

The hardware implementation for the binarized CorNET framework presented in Chapter 7 is the first circuit realization tailored for a recurrent neural network. Few works in the literature have addressed the hardware complexity inherent to LSTM layers. It involved binarization and quantization processes on the network model, which led to low degradation impact (< 1.2bpm) on the overall model accuracy when considering 22 recordings from the IEEE SPC dataset. The bCorNET framework is the first binary deep learning model for heart rate estimation available on the scientific community. Further, the proposed hardware architecture is suitable for embedded applications given the small

circuit footprint and low clock frequency required to operate on real-time HR estimation. The uniqueness of the proposed circuit is a significant contribution as it explores the implementation challenges of binary LSTMs, which were not subject to the due research, and it addresses the dataflow issues of stream architectures.

8.2 Future directions

The inherent robustness of neural networks makes them an excellent application for approximate computing techniques. In that sense, there is a plethora of methods and techniques that could be explored, especially in the context of recurrent neural networks where the hardware exploration has not received significant attention.

Also, many research projects require energy-efficient arithmetic operators as the current EDA tools do not always implement the best architecture. This issue could be solved with the RTLGen framework, which will integrate state-of-the-art arithmetic circuits that include approximate adders and multipliers. The framework can also be extended to generate complete arithmetic units in both fixed and floating points.

Finally, other architectural explorations can be performed on the bCorNET framework to improve the system accuracy and, if possible, reducing the overall complexity for optimal energy efficiency. These explorations may include adapting the architecture to consider additional input signals like accelerometers or multiple PPG channels, hyperparameter tuning, and layer and filter rearrangement. The framework can also be extended to measure multiple output signals like heart rate variability, blood pressure, blood oxygen saturation, among others.

REFERENCES

ABADI, M. et al. **TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems**. 2015. Software available from tensorflow.org. Available from Internet: <https://www.tensorflow.org/>.

ABDELGAWAD, A. Low power multiply accumulate unit (MAC) for future Wireless Sensor Networks. In: **2013 IEEE Sensors Applications Symposium Proceedings**. [S.l.: s.n.], 2013. p. 129–132.

ABDELGAWAD, A.; BAYOUMI, M. High Speed and Area-Efficient Multiply Accumulate (MAC) Unit for Digital Signal Processing Applications. In: **2007 IEEE International Symposium on Circuits and Systems**. [S.l.: s.n.], 2007. p. 3199–3202.

ACCELERA ORGANIZATION. **Universal Verification Methodology (UVM)**. 2012.

ALLEN, J. Photoplethysmography and its application in clinical physiological measurement. **Physiological Measurement**, v. 28, n. 3, p. R1–R39, mar 2007. ISSN 0967-3334. Available from Internet: <https://iopscience.iop.org/article/10.1088/0967-3334/28/3/R01>.

ANDERSON, R. R.; PARRISH, J. A. The optics of human skin. **Journal of Investigative Dermatology**, v. 77, n. 1, p. 13 – 19, 1981. ISSN 0022-202X. Available from Internet: <http://www.sciencedirect.com/science/article/pii/S0022202X15461251>.

ANDO, K. et al. BRein memory: A single-chip binary/ternary reconfigurable in-memory deep neural network accelerator achieving 1.4 TOPS at 0.6 w. **IEEE Journal of Solid-State Circuits**, Institute of Electrical and Electronics Engineers (IEEE), v. 53, n. 4, p. 983–994, apr 2018.

ANDRI, R. et al. YodaNN: An ultra-low power convolutional neural network accelerator based on binary weights. In: **2016 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)**. [S.l.]: IEEE, 2016.

AOYAGI, T.; MIYASAKA, K. Pulse oximetry: Its invention, contribution to medicine, and future tasks. **Anesthesia and analgesia**, v. 94, p. S1–3, 02 2002.

BANKMAN, D. et al. An always-on 3.8μJ/86% CIFAR-10 mixed-signal binary CNN processor with all memory on chip in 28nm CMOS. In: **2018 IEEE International Solid - State Circuits Conference - (ISSCC)**. [S.l.]: IEEE, 2018. p. 222–224.

BAUGH, C.; WOOLEY, B. A Two's Complement Parallel Array Multiplication Algorithm. **IEEE Transactions on Computers**, C-22, n. 12, p. 1045–1047, 1973. ISSN 0018-9340. Available from Internet: <http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=1672241>.

BEWICK, G. W. **Fast Multiplication : Algorithms and Implementation**. 170 p. Thesis (PhD) — Stanford University, 1994.

BISWAS, A.; CHANDRAKASAN, A. P. Conv-RAM: An energy-efficient SRAM with embedded convolution computation for low-power CNN-based machine learning applications. In: **2018 IEEE International Solid - State Circuits Conference - (ISSCC)**. [S.l.: s.n.], 2018. p. 488–490.

BISWAS, D. et al. Heart Rate Estimation From Wrist-Worn Photoplethysmography: A Review. **IEEE Sensors Journal**, p. 1–1, 2019.

BISWAS, D. et al. CorNET: Deep Learning Framework for PPG-Based Heart Rate Estimation and Biometric Identification in Ambulant Environment. **IEEE Transactions on Biomedical Circuits and Systems**, v. 13, n. 2, p. 282–291, April 2019.

BOOTH, A. D. A signed binary multiplication technique. **Quarterly Journal of Mechanics and Applied Mathematics**, v. 4, n. 2, p. 236–240, 1951.

BRENT, R. P. B.; KUNG, H. T. A Regular Layout for Parallel Adders. **IEEE Transactions on Computers**, C-31, n. 3, p. 260–264, 1982. ISSN 0018-9340.

BRUNIE, N. et al. Arithmetic core generation using bit heaps. In: **2013 23rd International Conference on Field programmable Logic and Applications**. [S.l.: s.n.], 2013. p. 1–8. ISSN 1946-147X.

CADENCE. **Cadence EDA tools**. 2018. Http://www.cadence.com.

CAVIGELLI, L. et al. Origami: A convolutional network accelerator. **CoRR**, abs/1512.04295, 2015.

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.

CHELLAPILLA, K.; PURI, S.; SIMARD, P. High Performance Convolutional Neural Networks for Document Processing. In: LORETTE, G. (Ed.). **Tenth International Workshop on Frontiers in Handwriting Recognition**. La Baule (France): Suvisoft, 2006. Available from Internet: <https://hal.inria.fr/inria-00112631>.

CHEN, T. et al. Diannao: A small-footprint high-throughput accelerator for ubiquitous machine-learning. In: **Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems**. New York, NY, USA: Association for Computing Machinery, 2014. (ASPLOS '14), p. 269–284.

CHEN, Y. et al. Diannao family: Energy-efficient hardware accelerators for machine learning. **Commun. ACM**, Association for Computing Machinery, New York, NY, USA, v. 59, n. 11, p. 105–112, oct. 2016.

CHEN, Y.-H.; EMER, J.; SZE, V. Eyeriss: A Spatial Architecture for Energy-Efficient Dataflow for Convolutional Neural Networks. In: **2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)**. [S.l.]: IEEE, 2016.

CHEN, Y.-h.; EMER, J.; SZE, V. Using Dataflow to Optimize Energy Efficiency of Deep Neural Network Accelerators. **IEEE Micro**, v. 37, n. 3, p. 12–21, 2017. ISSN 0272-1732. Available from Internet: <http://ieeexplore.ieee.org/document/7948671/>.

CHEN, Y.-H. et al. Eyeriss: An Energy-Efficient Reconfigurable Accelerator for Deep Convolutional Neural Networks. **IEEE Journal of Solid-State Circuits**, Institute of Electrical and Electronics Engineers (IEEE), v. 52, n. 1, p. 127–138, jan 2017.

CHETLUR, S. et al. cuDNN: Efficient Primitives for Deep Learning. **CoRR**, abs/1410.0759, 2014.

CHOLLET, F. et al. **Keras**. 2015. <https://keras.io>.

CHUNG, H.; LEE, H.; LEE, J. Finite State Machine Framework for Instantaneous Heart Rate Validation Using Wearable Photoplethysmography During Intensive Exercise. **IEEE J. Biomed. Health Inform.**, Institute of Electrical and Electronics Engineers (IEEE), v. 23, n. 4, p. 1595–1606, jul. 2019.

CONG, J.; XIAO, B. Minimizing computation in convolutional neural networks. In: WERMTER, S. et al. (Ed.). **Artificial Neural Networks and Machine Learning – ICANN 2014**. Cham: Springer International Publishing, 2014. p. 281–290. ISBN 978-3-319-11179-7.

CORNELISSE, D. **An intuitive guide to Convolutional Neural Networks**. 2018. <https://medium.freecodecamp.org/an-intuitive-guide-to-convolutional-neural-networks-260c2de0a050>. Accessed: 2018-12-01.

COSTA, E.; BAMPI, S.; MONTEIRO, J. A new architecture for signed radix-2m pure array multipliers. In: **Proceedings. IEEE International Conference on Computer Design: VLSI in Computers and Processors**. [S.l.: s.n.], 2002. p. 112–117. ISSN 1063-6404.

COURBARIAUX, M.; BENGIO, Y. Binarynet: Training deep neural networks with weights and activations constrained to +1 or -1. **CoRR**, abs/1602.02830, 2016.

COURBARIAUX, M.; BENGIO, Y.; DAVID, J. Low precision arithmetic for deep learning. **CoRR**, abs/1412.7024, 2014.

COURBARIAUX, M.; BENGIO, Y.; DAVID, J.-P. BinaryConnect: Training Deep Neural Networks with binary weights during propagations. In: CORTES, C. et al. (Ed.). **Advances in Neural Information Processing Systems 28**. Curran Associates, Inc., 2015. p. 3123–3131. Available from Internet: <http://papers.nips.cc/paper/5647-binaryconnect-training-deep-neural-networks-with-binary-weights-during-propagations.pdf>.

DADDA, L. Some Schemes for Parallel Multipliers. **Colloque sur l'Algèbre de Boole**, 1965.

DE SILVA, D. et al. Toward Intelligent Industrial Informatics: A Review of Current Developments and Future Directions of Artificial Intelligence in Industrial Applications. **IEEE Industrial Electronics Magazine**, v. 14, n. 2, p. 57–72, jun 2020. ISSN 1932-4529. Available from Internet: <https://ieeexplore.ieee.org/document/9127167/>.

DENG, B. L. et al. Model compression and hardware acceleration for neural networks: A comprehensive survey. **Proceedings of the IEEE**, v. 108, n. 4, p. 485–532, 2020.

DENNARD, R. H. Past progress and future challenges in LSI technology: From dram and scaling to ultra-low-power CMOS. **IEEE Solid-State Circuits Magazine**, v. 7, n. 2, p. 29–38, Spring 2015.

DESCHAMPS, J.-P.; BIOUL, G. J. A.; SUTTER, G. D. **Synthesis of Arithmetic Circuits (FPGA, ASIC and Embedded Systems)**. [S.l.]: John Wiley & Sons, Inc, 2006. 556 p.

DIAZ, K. M. et al. Fitbit®: An accurate and reliable device for wireless physical activity tracking. **International Journal of Cardiology**, Elsevier BV, v. 185, p. 138–140, abr. 2015. Available from Internet: <https://doi.org/10.1016/j.ijcard.2015.03.038>.

DIELEMAN, S. et al. **Lasagne: First release.** 2015. Available from Internet: <http://dx.doi.org/10.5281/zenodo.27878>.

DONAHUE, J. et al. Long-Term Recurrent Convolutional Networks for Visual Recognition and Description. **IEEE Transactions on Pattern Analysis and Machine Intelligence**, v. 39, n. 4, p. 677–691, apr 2017. ISSN 0162-8828. Available from Internet: <http://arxiv.org/abs/1411.4389http://ieeexplore.ieee.org/document/7558228/>.

DU, Z. et al. ShiDianNao: Shifting vision processing closer to the sensor. In: **2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA)**. [S.l.: s.n.], 2015. p. 92–104.

DU, Z. et al. Leveraging the error resilience of machine-learning applications for designing highly energy efficient accelerators. In: **2014 19th Asia and South Pacific Design Automation Conference (ASP-DAC)**. [S.l.: s.n.], 2014. p. 201–206.

DUCHI, J.; HAZAN, E.; SINGER, Y. Adaptive subgradient methods for online learning and stochastic optimization. **Journal of Machine Learning Research**, JMLR.org, v. 12, n. null, p. 2121–2159, jul. 2011. ISSN 1532-4435.

FARABET, C. et al. Hardware accelerated convolutional neural networks for synthetic vision systems. **Proceedings of 2010 IEEE International Symposium on Circuits and Systems**, p. 257–260, 2010.

FARABET, C. et al. CNP: An FPGA-based processor for Convolutional Networks. In: **2009 International Conference on Field Programmable Logic and Applications**. [S.l.: s.n.], 2009. p. 32–37.

FAROOQUI, A. A.; OKLOBDZIJA, V. G. General data-path organization of a MAC unit for VLSI implementation of DSP processors. **ISCAS '98. Proceedings of the 1998 IEEE International Symposium on Circuits and Systems (Cat. No.98CH36187)**, v. 2, p. 260–263, 1998.

FLORES, M. et al. P4 medicine: How systems medicine will transform the healthcare sector and society. **Personalized Medicine**, v. 10, n. 6, p. 565–576, 2013.

GARLAND, J.; GREGG, D. Low Complexity Multiply Accumulate Unit for Weight-Sharing Convolutional Neural Networks. **IEEE Computer Architecture Letters**, v. 16, n. 2, p. 132–135, July 2017.

GÉRON, A. **Hands-On Machine Learning with Scikit-Learn & Tensorflow**. [S.l.: s.n.], 2017. 572 p. ISBN 9781491962299.

GERS, F. A.; SCHMIDHUBER, J.; CUMMINS, F. Learning to forget: Continual prediction with lstm. **Neural Computation**, v. 12, n. 10, p. 2451–2471, 2000. Available from Internet: <https://doi.org/10.1162/089976600300015015>.

GIBSON, A.; PATTERSON, J. **Deep Learning: A Practitioner's Approach**. [S.l.]: O'Reilly, 2017.

GOODFELLOW, I.; BENGIO, Y.; COURVILLE, A. **Deep Learning**. [S.l.]: MIT Press, 2016. <http://www.deeplearningbook.org>.

GRAVES, A. Generating Sequences With Recurrent Neural Networks. **CoRR**, p. 1–43, 2013. Available from Internet: <http://arxiv.org/abs/1308.0850>.

GRAVES, A.; JAITLY, N. Towards end-to-end speech recognition with recurrent neural networks. In: **Proceedings of the 31st International Conference on International Conference on Machine Learning - Volume 32**. [S.l.]: JMLR.org, 2014. (ICML'14), p. II–1764–II–1772.

GUO, K. et al. Software-Hardware Codesign for Efficient Neural Network Acceleration. **IEEE Micro**, v. 37, n. 2, p. 18–25, Mar 2017. ISSN 0272-1732.

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**, Institute of Electrical and Electronics Engineers (IEEE), v. 37, n. 1, p. 35–47, jan 2018.

HAN, S. et al. EIE: efficient inference engine on compressed deep neural network. **CoRR**, abs/1602.01528, 2016. Available from Internet: <http://arxiv.org/abs/1602.01528>.

HAN, S.; MAO, H.; DALLY, W. J. Deep Compression: Compressing Deep Neural Networks with Pruning, Trained Quantization and Huffman Coding. p. 1–14, 2015.

HATAMIAN, M.; CASH, G. A 70-MHz 8-bit x 8-bit parallel pipelined multiplier in 2.5-μm CMOS. **IEEE Journal of Solid-State Circuits**, v. 21, n. 4, p. 505–513, 1986. ISSN 0018-9200.

HE, K. et al. Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification. **CoRR**, abs/1502.01852, 2015.

HINTON, G. E. et al. Improving neural networks by preventing co-adaptation of feature detectors. **CoRR**, abs/1207.0580, 2012.

HOCHREITER, S.; SCHMIDHUBER, J. Long Short-Term Memory. **Neural Computation**, v. 9, n. 8, p. 1735–1780, nov 1997. ISSN 0899-7667. Available from Internet: <http://www.mitpressjournals.org/doi/10.1162/neco.1997.9.8.1735>.

HOU, L.; YAO, Q.; KWOK, J. T. Loss-aware Binarization of Deep Networks. **CoRR**, abs/1611.01600, 2016.

HUBARA, I. et al. Quantized Neural Networks: Training Neural Networks with Low Precision Weights and Activations. **CoRR**, abs/1609.07061, 2016.

IOFFE, S.; SZEGEDY, C. Batch normalization: Accelerating deep network training by reducing internal covariate shift. **CoRR**, abs/1502.03167, 2015.

ISLAM, S. M. R. et al. The Internet of Things for Health Care: A Comprehensive Survey. **IEEE Access**, v. 3, p. 678–708, 2015. ISSN 2169-3536. Available from Internet: <http://www.embase.com/search/results?subaction=viewrecord{&}from=export{&}id=L72034289{\%}5Cnhttp://dx.doi.org/10.1111/ijs.12479http://ieeexplore.ieee.org/document>.

JARCHI, D.; CASSON, A. Description of a Database Containing Wrist PPG Signals Recorded during Physical Exercise with Both Accelerometer and Gyroscope Measures of Motion. **Data**, v. 2, n. 1, p. 13, dec 2016. ISSN 2306-5729. Available from Internet: <http://www.mdpi.com/2306-5729/2/1/1>.

JIA, Y. et al. Caffe: Convolutional architecture for fast feature embedding. **arXiv preprint arXiv:1408.5093**, 2014.

JOUPPI, N. P. et al. In-datacenter performance analysis of a tensor processing unit. In: **Proceedings of the 44th Annual International Symposium on Computer Architecture**. New York, NY, USA: Association for Computing Machinery, 2017. (ISCA '17), p. 1–12. ISBN 9781450348928. Available from Internet: <https://doi.org/10.1145/3079856.3080246>.

KATZ, R. H. **Contemporary logic design**. 1st. ed. Redwood City, CA, USA: Benjamin-Cummings Publishing Co., 1994. ISBN 0805327134.

KHUSHHAL, A. et al. Validity and reliability of the apple watch for measuring heart rate during exercise. **Sports Medicine International Open**, Georg Thieme Verlag KG, v. 1, n. 06, p. E206–E211, oct. 2017. Available from Internet: <https://doi.org/10.1055/s-0043-120195>.

KILANI, D. et al. Introduction to power management. In: ____. **Power Management for Wearable Electronic Devices**. Cham: Springer International Publishing, 2020. p. 1–13. ISBN 978-3-030-37884-4.

KIM, M.; SMARAGDIS, P. Bitwise neural networks. **CoRR**, abs/1601.06071, 2016. Available from Internet: <http://arxiv.org/abs/1601.06071>.

KINGMA, D. P.; BA, J. Adam: A Method for Stochastic Optimization. p. 1–15, 2014. ISSN 09252312. Available from Internet: <http://arxiv.org/abs/1412.6980>.

KNOWLES, S. A family of adders. **Proceedings 15th IEEE Symposium on Computer Arithmetic. ARITH-15 2001**, p. 277–284, 2001.

KOGGE, P. M.; STONE, H. S. A Parallel Algorithm for the Efficient Solution of a General Class of Recurrence Equations. **IEEE Transactions on Computers**, C-22, n. 8, p. 786–793, 1973. ISSN 0018-9340.

KONIJNENBURG, M. et al. A Multi(bio)sensor Acquisition System With Integrated Processor, Power Management, 8×8 LED Drivers, and Simultaneously Synchronized ECG, BIO-Z, GSR, and Two PPG Readouts. **IEEE Journal of Solid-State Circuits**, v. 51, n. 11, p. 2584–2595, Nov 2016. ISSN 1558-173X.

KRESTINSKAYA, O.; JAMES, A. P.; CHUA, L. O. Neuromemristive circuits for edge computing: A review. **IEEE Transactions on Neural Networks and Learning Systems**, v. 31, n. 1, p. 4–23, 2020.

KRIZHEVSKY, A.; SUTSKEVER, I.; HINTON, G. E. ImageNet Classification with Deep Convolutional Neural Networks. In: **Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 1**. USA: Curran Associates Inc., 2012. (NIPS'12), p. 1097–1105.

LECUN, Y.; BENGIO, Y.; HINTON, G. Deep learning. **Nature**, v. 521, n. 7553, p. 436–444, may 2015.

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

LECUN, Y. et al. Gradient-based learning applied to document recognition. **Proceedings of the IEEE**, v. 86, n. 11, p. 2278–2324, Nov 1998. ISSN 0018-9219.

LEE, H.; CHUNG, H.; LEE, J. Motion artifact cancellation in wearable photoplethysmography using gyroscope. **IEEE Sensors Journal**, v. 19, n. 3, p. 1166–1175, 2019.

LI, D. et al. Evaluating the Energy Efficiency of Deep Convolutional Neural Networks on CPUs and GPUs. In: **2016 IEEE International Conferences on Big Data and Cloud Computing (BDCloud), Social Computing and Networking (SocialCom), Sustainable Computing and Communications (SustainCom) (BDCloud-SocialCom-SustainCom)**. [S.l.: s.n.], 2016. p. 477–484.

LIANG, F. et al. Toward Edge-Based Deep Learning in Industrial Internet of Things. **IEEE Internet of Things Journal**, IEEE, v. 7, n. 5, p. 4329–4341, 2020. ISSN 23274662.

LIN, Y.; ZHANG, S.; SHANBHAG, N. R. Variation-tolerant architectures for convolutional neural networks in the near threshold voltage regime. In: **2016 IEEE International Workshop on Signal Processing Systems (SiPS)**. [S.l.: s.n.], 2016. p. 17–22.

MACSORLEY, O. High-Speed Arithmetic in Binary Computers. **Proceedings of the IRE**, v. 49, n. 1, 1961.

MARBLESTONE, A. H.; WAYNE, G.; KORDING, K. P. Toward an Integration of Deep Learning and Neuroscience. **Frontiers in Computational Neuroscience**, v. 10, p. 94, 2016. ISSN 1662-5188.

MARTINS, A.; FONSECA, M.; COSTA, E. Optimal combination of dedicated multiplication blocks and adder trees schemes for optimized radix-2^m array multipliers realization. In: **IEEE International Conference on Electronics, Circuits and Systems**. [S.l.: s.n.], 2015. p. 1–4.

MARTIS, R. J.; ACHARYA, U. R.; ADELI, H. Current methods in electrocardiogram characterization. **Computers in Biology and Medicine**, v. 48, p. 133 – 149, 2014. ISSN 0010-4825. Available from Internet: <http://www.sciencedirect.com/science/article/pii/S0010482514000432>.

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

MISRA, J.; SAHA, I. Artificial neural networks in hardware: A survey of two decades of progress. **Neurocomputing**, v. 74, n. 1, p. 239 – 255, 2010. Artificial Brains.

MITCHELL, T. M. **Machine Learning**. [S.l.]: McGraw-Hill Education, 1997. 432 p. ISSN 18684394. ISBN 0070428077.

MOONS, B.; BANKMAN, D.; VERHELST, M. **Embedded Deep Learning**. Cham: Springer International Publishing, 2019. ISBN 978-3-319-99222-8. Available from Internet: <http://link.springer.com/10.1007/978-3-319-99223-5>.

MOONS, B. et al. Minimum Energy Quantized Neural Networks. **CoRR**, abs/1711.00215, 2017.

MOONS, B.; VERHELST, M. An Energy-Efficient Precision-Scalable ConvNet Processor in 40-nm CMOS. **IEEE Journal of Solid-State Circuits**, Institute of Electrical and Electronics Engineers (IEEE), v. 52, n. 4, p. 903–914, apr 2017.

NAIR, V.; HINTON, G. E. Rectified Linear Units Improve Restricted Boltzmann Machines. In: **Proceedings of the 27th International Conference on International Conference on Machine Learning**. USA: Omnipress, 2010. (ICML'10), p. 807–814.

NURVITADHI, E. et al. Can FPGAs Beat GPUs in Accelerating Next-Generation Deep Neural Networks? In: **Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays**. New York, NY, USA: ACM, 2017. (FPGA '17), p. 5–14.

NVIDIA, C. Cublas library. **NVIDIA Corporation, Santa Clara, California**, v. 15, n. 27, p. 31, 2008.

OKLOBDZIJA, V. G.; KRISHNAMURTHY, R. K. **High-Performance Energy-Efficient Microprocessor Design**. [S.l.]: Springer US, 2006. 342 p.

OLAH, C. **Understanding LSTM Networks**. 2015. Accessed: 24/07/2020. Available from Internet: <https://colah.github.io/posts/2015-08-Understanding-LSTMs/>.

OLAH, C.; MORDVINTSEV, A.; SCHUBERT, L. Feature visualization. **Distill**, Distill Working Group, v. 2, n. 11, nov. 2017. Available from Internet: <https://doi.org/10.23915/distill.00007>.

PAIM, G. et al. Power-, area-, and compression-efficient eight-point approximate 2-d discrete tchebichef transform hardware design combining truncation pruning and efficient transposition buffers. **IEEE Transactions on Circuits and Systems I: Regular Papers**, v. 66, p. 680–693, 2019.

PAIM, G. et al. Power-, area-, and compression-efficient eight-point approximate 2-d discrete tchebichef transform hardware design combining truncation pruning and efficient transposition buffers. **IEEE Transactions on Circuits and Systems I: Regular Papers**, v. 66, n. 2, p. 680–693, 2019.

PASZKE, A. et al. Pytorch: An imperative style, high-performance deep learning library. In: WALLACH, H. et al. (Ed.). **Advances in Neural Information Processing Systems 32**. Curran Associates, Inc., 2019. p. 8024–8035. Available from Internet: <http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf>.

PATTERSON, D. A.; HENNESSY, J. L. **Computer Organization and Design: The Hardware/Software Interface**. 5th. ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2013. ISBN 0124077269.

PIEPER, L.; COSTA, E.; MONTEIRO, J. Combination of radix-2^m multiplier blocks and adder compressors for the design of efficient 2's complement 64-bit array multipliers. In: **26th Symposium on Integrated Circuits and Systems Design**. [S.l.: s.n.], 2013. p. 1–6.

PURWINS, H. et al. Deep Learning for Audio Signal Processing. **IEEE Journal of Selected Topics in Signal Processing**, v. 13, n. 2, p. 206–219, may 2019. ISSN 1932-4553. Available from Internet: <https://ieeexplore.ieee.org/document/8678825/>.

RAJENDRAN, B.; ALIBART, F. Neuromorphic computing based on emerging memory technologies. **IEEE Journal on Emerging and Selected Topics in Circuits and Systems**, v. 6, n. 2, p. 198–211, 2016.

RASTEGARI, M. et al. XNOR-Net: ImageNet Classification Using Binary Convolutional Neural Networks. **CoRR**, abs/1603.05279, 2016.

ROCHA, L. G. et al. Real-time HR Estimation from wrist PPG using Binary LSTMs. In: **2019 IEEE Biomedical Circuits and Systems Conference (BioCAS)**. [S.l.: s.n.], 2019. p. 1–4. ISSN 2163-4025.

ROCHA, L. M. G. et al. Binary CorNET: Accelerator for HR estimation from wrist-PPG. **IEEE Transactions on Biomedical Circuits and Systems**, p. 1–1, 2020. ISSN 1940-9990.

ROCHA, L. M. G. et al. Framework-based arithmetic core generation to explore ASIC-based parallel binary multipliers. In: **2017 24th IEEE International Conference on Electronics, Circuits and Systems (ICECS)**. IEEE, 2017. v. 2018-January, p. 478–481. ISBN 978-1-5386-1911-7. Available from Internet: <http://ieeexplore.ieee.org/document/8292065/>.

ROSENBLATT, F. The perceptron: A probabilistic model for information storage and organization in the brain. **Psychological Review**, American Psychological Association (APA), v. 65, n. 6, p. 386–408, 1958.

ROUHOLAMINI, M. et al. A New Design for 7:2 Compressors. In: **2007 IEEE/ACS International Conference on Computer Systems and Applications**. [S.l.: s.n.], 2007. p. 474–478.

RUSSAKOVSKY, O. et al. ImageNet Large Scale Visual Recognition Challenge. **International Journal of Computer Vision (IJCV)**, v. 115, n. 3, p. 211–252, 2015.

RYBALKIN, V. et al. Hardware Architecture of Bidirectional Long Short-term Memory Neural Network for Optical Character Recognition. In: **Proceedings of the Conference on Design, Automation & Test in Europe**. 3001 Leuven, Belgium, Belgium: European Design and Automation Association, 2017. (DATE '17), p. 1394–1399.

SAK, H.; SENIOR, A.; BEAUFAYS, F. Long Short-Term Memory Based Recurrent Neural Network Architectures for Large Vocabulary Speech Recognition. **CoRR**, 2014. Available from Internet: <http://arxiv.org/abs/1402.1128>.

SATIJA, U.; RAMKUMAR, B.; MANIKANDAN, S. M. Real-Time Signal Quality-Aware ECG Telemetry System for IoT-Based Health Care Monitoring. **IEEE Internet of Things Journal**, v. 4, n. 3, p. 815–823, 2017.

SCHONLE, P. et al. A multi-sensor and parallel processing SoC for wearable and implantable telemetry systems. In: **ESSCIRC 2017 - 43rd IEEE European Solid State Circuits Conference**. [S.l.: s.n.], 2017. p. 215–218. ISSN null.

SCHWIERZ, F.; LIOU, J.; WONG, H. **Nanometer CMOS**. [S.l.]: Pan Stanford, 2010. ISBN 9789814241083.

SESHADRI, D. R. et al. Wearable sensors for monitoring the internal and external workload of the athlete. **npj Digital Medicine**, Springer Science and Business Media LLC, v. 2, n. 1, jul. 2019. Available from Internet: <https://doi.org/10.1038/s41746-019-0149-2>.

SILVEIRA, B. et al. Power-efficient sum of absolute differences hardware architecture using adder compressors for integer motion estimation design. **IEEE Transactions on Circuits and Systems I: Regular Papers**, PP, n. 99, p. 1–12, 2017. ISSN 1549-8328.

SJÄLANDER, M.; LARSSON-EDEFORS, P. **The Case for HPM-Based Baugh-Wooley Multipliers**. Göteborg, Sweden, 2008. 16 p.

SÖRNMO, L.; LAGUNA, P. **Bioelectrical Signal Processing in Cardiac and Neurological Applications**. Elsevier Science, 2005. (Academic Press series in biomedical engineering). ISBN 9780124375529. Available from Internet: <https://books.google.com.br/books?id=RQv7tFFXYyIC>.

STEINKRAU, D.; SIMARD, P. Y.; BUCK, I. Using GPUs for Machine Learning Algorithms. In: **Proceedings of the Eighth International Conference on Document Analysis and Recognition**. Washington, DC, USA: IEEE Computer Society, 2005. (ICDAR '05), p. 1115–1119.

STELLING, P. F.; OKLOBDZIJA, V. G. Implementing multiply-accumulate operation in multiplication time. In: **Proceedings 13th IEEE Sympsoium on Computer Arithmetic**. [S.l.: s.n.], 1997. p. 99–106.

STÖTTNER, T. **Why Data should be Normalized before Training a Neural Network**. Towards Data Science, 2019. Available from Internet: <https://towardsdatascience.com/why-data-should-be-normalized-before-training-a-neural-network-c626b7f66c7d>.

STRATH, S. J. et al. Evaluation of heart rate as a method for assessing moderate intensity physical activity. **Medicine & Science in Sports & Exercise**, Ovid Technologies (Wolters Kluwer Health), v. 32, n. Supplement, p. S465–S470, sep. 2000. Available from Internet: <https://doi.org/10.1097/00005768-200009001-00005>.

SUN, J. et al. Attenuation of marine seismic interference noise employing a customized u-net. **Geophysical Prospecting**, v. 68, n. 3, p. 845–871, 2020. Available from Internet: <https://onlinelibrary.wiley.com/doi/abs/10.1111/1365-2478.12893>.

SUN, Y.; THAKOR, N. Photoplethysmography revisited: From contact to noncontact, from point to imaging. **IEEE Transactions on Biomedical Engineering**, v. 63, n. 3, p. 463–477, 2016.

SUTSKEVER, I.; VINYALS, O.; LE, Q. V. Sequence to sequence learning with neural networks. In: GHAHRAMANI, Z. et al. (Ed.). **Advances in Neural Information Processing Systems 27**. Curran Associates, Inc., 2014. p. 3104–3112. Available from Internet: <http://papers.nips.cc/paper/5346-sequence-to-sequence-learning-with-neural-networks.pdf>.

SZE, V. et al. Hardware for machine learning: Challenges and opportunities. In: **2017 IEEE Custom Integrated Circuits Conference (CICC)**. [S.l.]: IEEE, 2017. p. 1–8.

SZE, V. et al. Efficient Processing of Deep Neural Networks: A Tutorial and Survey. **Proceedings of the IEEE**, Institute of Electrical and Electronics Engineers (IEEE), v. 105, n. 12, p. 2295–2329, dec 2017.

TAMURA, T. et al. Wearable Photoplethysmographic Sensors—Past and Present. **Electronics**, v. 3, n. 2, p. 282–302, 2014. Available from Internet: <https://www.mdpi.com/2079-9292/3/2/282>.

TEMKO, A. Accurate Heart Rate Monitoring During Physical Exercises Using PPG. **IEEE Trans. Biomed. Eng.**, v. 64, n. 9, p. 2016–2024, Sep. 2017. ISSN 0018-9294.

THEANO DEVELOPMENT TEAM. Theano: A Python framework for fast computation of mathematical expressions. **arXiv e-prints**, may 2016. Available from Internet: <http://arxiv.org/abs/1605.02688>.

TIMMONS, N. G.; RICE, A. **Approximating Activation Functions**. 2020.

TYAGI, A. A reduced-area scheme for carry-select adders. **IEEE Transactions on Computers**, v. 42, n. 10, p. 1163–1170, 1993. ISSN 0018-9340.

VANHOUCKE, V.; SENIOR, A.; MAO, M. Z. Improving the speed of neural networks on CPUs. In: CITESEER. **Proc. Deep Learning and Unsupervised Feature Learning NIPS Workshop**. [S.l.], 2011. v. 1, p. 4.

VELASCO-MONTERO, D. et al. Optimum Selection of DNN Model and Framework for Edge Inference. **IEEE Access**, v. 6, p. 51680–51692, 2018. ISSN 21693536.

VERHELST, M.; MOONS, B. Embedded Deep Neural Network Processing: Algorithmic and Processor Techniques Bring Deep Learning to IoT and Edge Devices. **IEEE Solid-State Circuits Magazine**, v. 9, n. 4, p. 55–65, 2017. ISSN 1943-0582.

WALLACE, C. S. A Suggestion for a Fast Multiplier. **IEEE Transactions on Electronic Computers**, EC-13, n. 1, p. 14–17, 1964.

WASON, R. Deep learning: Evolution and expansion. **Cognitive Systems Research**, v. 52, p. 701 – 708, 2018. ISSN 1389-0417. Available from Internet: <http://www.sciencedirect.com/science/article/pii/S1389041717303546>.

WEI, X. et al. Automated Systolic Array Architecture Synthesis for High Throughput CNN Inference on FPGAs. In: **Proceedings of the 54th Annual Design Automation Conference 2017**. [S.l.]: Association for Computing Machinery, 2017. (DAC '17).

WEINBERGER, A. **4-2 Carry-Save Adder Module**. [S.l.], 1981.

WIDROW, B.; LEHR, M. A. 30 years of adaptive neural networks: perceptron, Madaline, and backpropagation. **Proceedings of the IEEE**, v. 78, n. 9, p. 1415–1442, Sept 1990. ISSN 0018-9219.

XIONG, W. et al. **Achieving Human Parity in Conversational Speech Recognition**. 2016.

YANG, G. et al. IoT-Based Remote Pain Monitoring System: From Device to Cloud Platform. **IEEE Journal of Biomedical and Health Informatics**, v. 22, n. 6, p. 1711–1719, 2018.

YANG, T.; CHEN, Y.; SZE, V. Designing energy-efficient convolutional neural networks using energy-aware pruning. In: **2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)**. [S.l.: s.n.], 2017. p. 6071–6079.

YU, D. et al. Mixed pooling for convolutional neural networks. In: MIAO, D. et al. (Ed.). **Rough Sets and Knowledge Technology**. Cham: Springer International Publishing, 2014. p. 364–375. ISBN 978-3-319-11740-9.

ZHANG, Q. et al. ApproxANN: An Approximate Computing Framework for Artificial Neural Network. In: **Design, Automation & Test in Europe Conference & Exhibition (DATE), 2015**. [S.l.: s.n.], 2015.

ZHANG, Z. Photoplethysmography-Based Heart Rate Monitoring in Physical Activities via Joint Sparse Spectrum Reconstruction. **IEEE Transactions on Biomedical Engineering**, v. 62, n. 8, p. 1902–1910, aug 2015. ISSN 0018-9294. Available from Internet: <http://ieeexplore.ieee.org/document/7047715/>.

ZHANG, Z.; PI, Z.; LIU, B. TROIKA: A General Framework for Heart Rate Monitoring Using Wrist-Type Photoplethysmographic Signals During Intensive Physical Exercise. **IEEE Transactions on Biomedical Engineering**, v. 62, n. 2, p. 522–531, Feb 2015. ISSN 0018-9294.

ZHOU, S. et al. DoReFa-Net: Training Low Bitwidth Convolutional Neural Networks with Low Bitwidth Gradients. **CoRR**, abs/1606.06160, 2016.

ZHOU, Z. et al. Edge Intelligence: Paving the Last Mile of Artificial Intelligence With Edge Computing. **Proceedings of the IEEE**, v. 107, n. 8, 2019. ISSN 00189219.

ZIPSER, D.; WILLIAMS, R. J. Gradient-Based Learning Algorithms for Recurrent Networks and Their Computational Complexity. In: CHAUVIN, Y.; RUMELHART, D. E. (Ed.). **Back-propagation: Theory, Architectures and Applications**. [S.l.]: L. Erlbaum Associates Inc., 1995. p. 433–486. ISBN 0805812598.

APPENDIX A — PAPERS PUBLISHED DURING PH.D. PROGRAM

BOOK CHAPTERS

1. PAIM, Guilherme; SOARES, Leonardo B.; **ROCHA, Leandro M. G.**; ABREU, Brunno; SANTANA, Gustavo M.; DINIZ, Claudio M.; DA COSTA, Eduardo A.; BAMPI, Sergio: *Low-power circuit design techniques for high-resolution video coding*. VLSI Architectures for Future Video Coding. 1ed.: Institution of Engineering and Technology, 2019, p. 149-190.

PAPERS PUBLISHED OR SUBMITTED TO JOURNALS

1. **ROCHA, Leandro M. G.**; PAIM, Guilherme; SANTANA, Gustavo M.; COSTA, Eduardo A. C., BAMPI, Sergio.*Framework-based Arithmetic Datapath Generation to Explore Parallel Binary Multipliers*. Journal of Integrated Circuits and Systems, 2019. **(Under Peer Review)**
2. **ROCHA, Leandro M. G.**; BISWAS, Dwaipayan; VERHOEF, Bram; BAMPI, Sergio; VAN HOOF, Chris; KONIJNENBURG, Mario; VERHELST, Marian; VAN HELLEPUTTE, Nick. B*inary CorNET: Accelerator for HR estimation from wrist-PPG*. IEEE Transactions on Biomedical Circuits and Systems, 2020.
3. PAIM, Guilherme P.; SANTANA, Gustavo M.; ABREU, Brunno A.; **ROCHA, Leandro M. G.**; GRELLERT, Mateus; COSTA, Eduardo A. C.; BAMPI, Sergio. *Exploring High-Order Adder Compressors for Reducing Power in Sum of Absolute Differences Architectures for Real-time UHD Video Encoding*. Journal of Real-Time Image Processing, v. 1, p. 1-1, 2020.
4. GUIDOTTI, Vagner; PAIM, Guilherme; **ROCHA, Leandro M. G.**; COSTA, Eduardo A. C.; ALMEIDA, Sérgio; BAMPI, Sergio. *Power-Efficient Approximate Newton-Raphson Integer Divider Applied to NLMS Adaptive Filter for High-Quality Interference Cancelling*. Circuits, Systems and Signal Processing, v. 39, p. 1-1, 2020.
5. LIMA, Vitor G.; WUERDIG, Rodrigo; PAIM, Guilherme P.; **ROCHA, Leandro M. G.**; ROSA Jr, Leomar; MARQUES, Felipe; VALDUGA, Vinícius C.; COSTA,

Eduardo A. C.; SOARES, Rafael I.; BAMPI, Sergio. *Enhancing Side Channel Attack-Resistance of the STTL Combining Multi-Vt Transistors with Capacitance and Current Paths Counterbalancing*. Journal of Integrated Circuits and Systems, v. 15, p. 1, 2020.

6. ABREU, Brunno A.; GRELLERT, Mateus; PAIM, Guilherme; ROCHA, **Leandro M. G.**; DINIZ, Claudio M.; COSTA, Eduardo A. C.; BAMPI, Sergio. *Exploring Absolute Differences Arithmetic Operators for Power- and Area-Efficient SAD Hardware Architectures*. Journal of Integrated Circuits and Systems, v. 15, p. 1, 2020.
7. FONTANARI, Thomas; PAIM, Guilherme; **ROCHA, Leandro M. G.**; *SANTANA, Gustavo M.*; COSTA, Eduardo A. C.; BAMPI, Sergio. *A Fast Monolithic 8-2 Adder Compressor Circuit*. Journal of Integrated Circuits and Systems, v. 14, p. 1-7, 2019.
8. PAIM, Guilherme; **ROCHA, Leandro M. G.**; AMROUCH, Hussam; COSTA, Eduardo A. C.; BAMPI, Sergio; HENKEL, Jorg. *A Cross-layer Gate-Level-to-Application Co-simulation for Design Space Exploration of Approximate Circuits in HEVC Video Encoders*. IEEE Transactions on Circuits and Systems for Video Technology, v. 1, p. 1-1, 2019.
9. FONTANARI, Thomas; **ROCHA, Leandro M. G.**; SANTANA, Gustavo M.; PAIM, Guilherme P.; COSTA, Eduardo A. C.; BAMPI, Serio. *Investigating the Impact of Pruning on Energy-Accuracy of Neural Networks*. Revista Júnior de Iniciação Científica em Ciências Exatas e Engenharia, v. 20, p. 1, 2019.
10. PAIM, Guilherme; **ROCHA, Leandro M. G.**; SANTANA, Gustavo; SOARES, Leonardo; COSTA, Eduardo A. C.; BAMPI, Sergio. *Using Pruning and Truncation for Power-Efficient 2-D Approximate Tchebichef Transform Hardware Architecture*. Journal of Integrated Circuits and Systems, v. 13, p. 1-6, 2018.
11. PAIM, Guilherme; **ROCHA, Leandro M. G.**; SANTANA, Gustavo M.; SOARES, Leonardo B.; COSTA, Eduardo A. C.; BAMPI, Sergio. *Power-, Area-, and Compression-Efficient Eight-Point Approximate 2-D Discrete Tchebichef Transform Hardware Design Combining Truncation Pruning and Efficient Transposition Buffers*. IEEE Transactions on Circuits and Systems I: Regular Papers, v. 1, p. 1-14, 2018.
12. PAIM, Guilherme; SANTANA, Gustavo M.; **ROCHA, Leandro M. G.**; SOARES, Leonardo B.; COSTA, Eduardo A. C.; BAMPI, Sergio. *Exploring approximations in 4- and 8- point DTT hardware architectures for low-power image compression*.

Analog Integrated Circuits and Signal Processing, v. 1, p. 1-1, 2018.

PAPERS PUBLISHED TO CONFERENCES

1. FERREIRA, Guilherme; **ROCHA, Leandro M. G.**; COSTA, Eduardo A. C.; BAMPI, Sergio. *Combining m=2 Multipliers and Adder Compressors for Power Efficient Radix-4 Butterfly*. In: Midwest Symposium on Circuits and Systems, 2020, Springfield. MWSCAS'2020, 2020.
2. LEME, Mateus T.; PAIM, Guilherme P.; **ROCHA, Leandro M. G.**; UCKER, Patrícia; LIMA, Vitor G.; SOARES, Rafael I.; COSTA, Eduardo A. C.; BAMPI, Sergio. *Optimizing the Montgomery Modular Multiplier for a Power- and Area-Efficient Hardware Architecture*. In: Midwest Symposium on Circuits and Systems, 2020, Springfield. MWSCAS'2020, 2020.
3. **ROCHA, Leandro M. G.**; ROSA, Morgana; PAIM, Guilherme P.; COSTA, Eduardo A. C.; BAMPI, Sergio. *Improving the Partial Product Tree Compression on Signed Radix-2^m Parallel Multipliers*. In: 18th IEEE International New Circuits and Systems, 2020, Montreal. NEWCAS'2020, 2020. **(Invited for TCAS-I)**
4. FONTANARI, Thomas; PAIM, Guilherme; **ROCHA, Leandro M. G.**; UCKER, P.; COSTA, Eduardo A. C.; BAMPI, Sergio. *An Efficient N-bit 8-2 Adder Compressor with a Constant Internal Carry Propagation Delay*. In: An Efficient N-bit 8-2 Adder Compressor with a Constant Internal Carry Propagation Delay, 2020, San José. LASCAS 2020, 2020.
5. **ROCHA, Leandro M. G.**; LIU, Muqing; BISWAS, Dwaipayan; VERHOEF, Bram; BAMPI, Sergio; KIM, Chris H.; HOOF, Chris V.; KONIJNENBURG, Mario; VERHELST, Marian; VAN HELLEPUTTE, Nick. *Real-time HR Estimation from wrist PPG using Binary LSTMs*. In: IEEE Biomedical Circuits and Systems Conference, 2019, Nara. BIOCAS'19, 2019.
6. LIMA, V. G.; PAIM, Guilherme P.; **ROCHA, Leandro M. G.**; ROSA Jr, Leomar; MARQUES, Felipe; COSTA, Eduardo A. C.; VALDUGA, Vinícius C.; SOARES, Rafael I.; BAMPI, Sergio. *Maximizing Side Channel Attack-Resistance and Energy-Efficiency of the STTL Combining Multi-Vt Transistors with Current and Capacitance Balancing*. In: IEEE International Symposium on Circuits and Systems,

2019, Sapporo. ISCAS'19, 2019.

7. PAIM, Guilherme; **ROCHA, Leandro M. G.**; COSTA, Eduardo A. C.; BAMPI, Sergio. *Maximizing the Power-Efficiency of the Approximate Pruned Modified Rounded DCT Exploiting Approximate Adder Compressors.* In: IEEE International New Circuits and Systems Conference, 2019, Munich. NEWCAS'19, 2019.
8. ABREU, Brunno; GRELLERT, M.; PAIM, Guilherme P.; FONTANARI, Thomas; **ROCHA, Leandro M. G.**; COSTA, Eduardo A. C.; BAMPI, Sergio. *Exploring Motion Vector Cost with Partial Distortion Elimination in Sum of Absolute Differences for HEVC Integer Motion Estimation.* In: IEEE International New Circuits and Systems Conference, 2019, Munich. NEWCAS'19, 2019.
9. ABREU, Brunno; SANTANA, Gustavo; GRELLERT, Mateus; PAIM, Guilherme; **ROCHA, Leandro M. G.**; DA COSTA, Eduardo A. C.; BAMPI, Sergio. *Exploiting Partial Distortion Elimination in the Sum of Absolute Differences for Energy-Efficient HEVC Integer Motion Estimation*. In: 2018 31st Symposium on Integrated Circuits and Systems Design (SBCCI), Bento Gonçalves, 2018. p. 1.
10. SEQUEIRA, Luis F.; SANTANA, Gustavo M.; PAIM, Guilherme; **ROCHA, Leandro M. G.**; ABREU, Brunno; COSTA, Eduardo; BAMPI, Sergio. *Low-Power HEVC 8-point 2-D Discrete Cosine Transform Hardware Using Adder Compressors*. In: 2018 16th IEEE International New Circuits and Systems Conference (NEWCAS), 2018, Montreal.
11. ALVES, Tiago G.; PAIM, Guilherme P.; **ROCHA, Leandro M. G.**; FERREIRA, R.; HENRIQUES, R. V. B.; COSTA, Eduardo A. C.; BAMPI, Sergio. *A Power-Predictive Environment for Fast and Power-Aware ASIC-Based FIR Filter Design*. In: Symposium on Integrated Circuits and Systems Design (SBCCI), 2017, Fortaleza. SBCCI Proceedings, 2017.
12. **ROCHA, Leandro M. G.**; PAIM, Guilherme P.; COSTA, Eduardo A. C.; BAMPI, Sergio. *Impact of SRAM IP Aspect Ratio in ASIC-Oriented Viterbi Decoder Physical Implementation*. In: Workshop on Circuits and System Design, 2017, Fortaleza. WCAS Proceedings, 2017.
13. **ROCHA, Leandro M. G.**; PAIM, Guilherme P.; SANTANA, Gustavo M.; ABREU, Brunno A.; FERREIRA, Rafael; COSTA, Eduardo A. C.; BAMPI, Sergio. *Physical Implementation of an ASIC-Oriented SRAM-Based Viterbi Decoder*. In: 24th IEEE International Conference on Electronics, Circuits and Systems, 2017, Batumi.

ICECS Proceedings, 2017.

14. **ROCHA, Leandro M. G.;** PAIM, Guilherme P.; FERREIRA, R.; COSTA, Eduardo A. C.; BAMPI, Sergio. *Framework-based Arithmetic Core Generation to Explore ASIC-based Parallel Binary Multipliers*. In: 24th IEEE International Conference on Electronics, Circuits and Systems, 2017, Batumi. ICECS Proceedings, 2017.
15. SANTANA, Gustavo M.; PAIM, Guilherme P.; **ROCHA, Leandro M. G.**; NEUENFELD, Renato; FONSECA, Mateus B.; COSTA, Eduardo A. C.; BAMPI, Sergio. *Using Efficient Adder Compressors with a Split-Radix Butterfly Hardware Architecture for Low-Power IoT Smart Sensors*. In: 24th IEEE International Conference on Electronics, Circuits and Systems, 2017, Batumi. ICECS Proceedings, 2017.