THOMAS ARAUJO MUYAL

OpenNPU: an open source platform for automatic neural network synthesis for FPGAs

Thesis submitted for the degree of Master in Science to the Escola Politécnica of Universidade de São Paulo.

São Paulo
2023

THOMAS ARAUJO MUYAL

OpenNPU: an open source platform for automatic neural network synthesis for FPGAs

Versão corrigida

Thesis submitted for the degree of Master in Science to the Escola Politécnica of Universidade de São Paulo.

Concentration field:

Systems Engineering

Advisor:

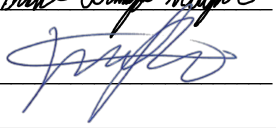
Prof. Dr. Marcelo Knörich Zuffo

São Paulo
2023

Este exemplar foi revisado e corrigido em relação à versão original, sob responsabilidade única do autor e com a anuência de seu orientador.

São Paulo, 16 de fevereiro de 2023

Assinatura do autor: 

Assinatura do orientador: 

Catalogação-na-publicação

Muyal, Thomas Araujo
OpenNPU: an open source platform for automatic neural network synthesis for FPGAs / T. A. Muyal -- versão corr. -- São Paulo, 2023.
118 p.

Dissertação (Mestrado) - Escola Politécnica da Universidade de São Paulo. Departamento de Engenharia de Sistemas Eletrônicos.

1.Inteligência artificial 2.Eletrônica digital 3.Circuitos integrados VLSI 4.Redes neurais 5.Internet das coisas I.Universidade de São Paulo. Escola Politécnica. Departamento de Engenharia de Sistemas Eletrônicos II.t.

RESUMO

Redes neurais artificiais são técnicas de inteligência artificial amplamente utilizadas na indústria atualmente, implementadas para a solução de uma grande variedade de problemas. Sua recente popularidade é em parte explicada pelo avanço de tecnologias de hardware, que fornecem recursos computacionais para o processamento dos seus cálculos. Software geralmente é executado em Unidades de Processamento Central (CPUs), de propósito geral, mas para a melhoria de métricas como performance e eficiência energética, utilizam-se aceleradores em hardware especializados. No contexto de computação de borda no âmbito da Internet das Coisas, estas restrições de hardware são ainda mais rigorosas. Uma abordagem para resolver o ambiente restritivo de computação de borda é a quantização de redes neurais, que consiste na redução da precisão de representação dos seus parâmetros para que consumam menos memória, e operações sejam mais rápidas e menos custosas. Um dos dispositivos utilizados para este propósito são Field-Programmable Gate Arrays (FPGAs), que oferecem circuitos integrados cuja funcionalidade pode ser programada para a implementação especializada de algoritmos, fornecendo alto desempenho e eficiência de execução, juntamente de benefícios como reconfigurabilidade, ciclos de design e time-to-market mais curtos, e custo mais baixo que alternativas. Um dos problemas para o uso de FPGAs é que sua programação é difícil e requer conhecimento especializado para aproveitamento de suas características, e nem toda equipe de desenvolvimento de inteligência artificial o detêm. Assim, há interesse em sistemas que automatizem a síntese de aceleradores de redes neurais em FPGAs, para trazer a maiores públicos as vantagens desta técnica. Este trabalho estuda o estado da arte deste tipo de software, estudando lacunas na área. O objetivo principal desta pesquisa é a criação e validação de uma prova de conceito de geração automática de aceleradores de hardware para redes neurais utilizando técnicas de quantização para possibilitar sua síntese em FPGAs pequenas, feitas para computação de borda. Para a validação deste sistema, aceleradores foram gerados e seus comportamentos foram testados nas métricas de latência, vazão de resultados, eficiência energética e área de circuito, e comparada com sua execução em uma CPU de computador pessoal. Os resultados indicam que os aceleradores gerados são sintetizáveis, significativamente mais rápidos e eficientes energéticamente que a implementação em CPU.

Palavras chave: Inteligência artificial, Aprendizagem de máquina, Redes neurais, Aprendizagem profunda, Acelerador em hardware, Internet das Coisas, FPGA.

ABSTRACT

Artificial neural networks are a group of artificial intelligence algorithms widely used contemporarily in the industry and research, implemented to solve a great variety of issues. Its recent popularity is in part due to the advance of hardware technologies, which provide computational resources for the resource-intensive processing necessary for its implementation. Historically, neural network software is executed in Central Processing Units (CPUs), which are general purpose devices. However, metrics such as inference speed, energy efficiency and circuit area are improved with the use of specialized hardware accelerators. In the context of edge computing, the processing on location of data gathered by Internet of Things (IoT) devices, there are significant restrictions as to those metrics. One of the devices frequently used for this purpose are Field-Programmable Gate Arrays (FPGAs), which offer integrated circuits whose functionality may be programmed for the synthesis of hardware specialized in specific algorithms, offering high performance and execution efficiency, as well as other benefits such as being able to be reconfigured after manufacturing, shorter design cycles, faster time-to-market and lower cost than the alternatives. One of the issues for the use of FPGAs is that its programming is difficult and requires specialist knowledge to fully make use of these devices' characteristics, and not every team of artificial intelligence developers has such knowledge. Consequently, there is interest in systems that automatize the synthesis of neural network hardware accelerators in FPGAs, to bring the benefits of this technique to a wider audience. Another approach to solve the high restriction environment of edge computing is neural network quantization, which means reducing the precision of representation of parameters so they consume less memory, and operations using these numbers are faster. This work studies the state of the art of this manner of software, diagnosing existing gaps. The main objective of this research is the creation and validation of a proof of concept of automatic generation of neural network hardware accelerators that are using parameter quantization techniques to enable its synthesis on small FPGAs aimed towards edge computing. For the validation of this system, an accelerator was generated and its behavior was measured in metrics of latency, end result throughput, energy efficiency, circuit area and compared to the execution of the same neural network in a high-end personal computer CPU. The results indicate that the generated hardware accelerator is synthesizeable, significantly faster and consumes considerably less energy than the CPU implementation.

Keywords: Artificial intelligence, Machine learning, Neural networks, Deep learning, Hardware accelerator, Internet of Things, FPGA.

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVIATIONS

AI	Artificial Intelligence
ASIC	Application-Specific Integrated Circuit
CPU	Central Processing Unit
FPGA	Field-Programmable Gate Array
GPGPU	General Purpose Graphics Processing Unit
GPU	Graphics Processing Unit
HDL	Hardware Description Language
ONNX	Open Neural Network Exchange
USP	Universidade de São Paulo
VHDL	VHSIC Hardware Description Language
VHSIC	Very High Speed Integrated Circuit

CONTENTS

1 INTRODUCTION

Presently, artificial neural networks have been successfully implemented in a vast array of different issues. For its adequate use, its hardware demands must be considered to obtain adequate solutions for each problem. Hardware demands concerning the computational needs to develop and implement artificial neural networks are divided in two phases: first, there is the development and training of the neural network, which consists in the design of its topology and algorithms, and training during which the artificial intelligence learns to execute its task. This training phase can be significantly resource-intensive in computational resources, and there are many concerns over accelerating and optimizing this step. Finally, once the network is considered sufficiently trained, the system can be deployed to execute its task. This phase also has considerations with its hardware implementation, as the final artificial neural network system architecture can be significantly resource intensive.

This work focuses on the final implementation of artificial neural networks which have already been trained.

The choice of architecture for their implementation is of crucial importance, as they must consider parameters such as execution speed, energy efficiency, circuit area usage, system reconfigurability, cost and design cycle complexity. This hardware choice must balance and account for these metrics, according to each application's specific demands. In general, the hardware choice lies between Central Processing Units (CPUs), Graphic Processing Units (GPUs), Application-Specific Integrated Circuits (ASICs) or Field-Programmable Gate Arrays (FPGAs).

Software in general is usually executed in either CPUs or GPUs, but for specific applications, their low energy efficiency discourages their use. ASICs, on the other hand, offer high performance and efficiency through fully customized and specialized hardware for its application. However, this alternative is also accompanied by difficulties in its design cycles, having high initial cost and low reconfigurability. (MITTAL, 2020)

FPGAs, meanwhile, offer better low power energy efficiency than CPUs and GPUs (MITTAL; VETTER, 2014). Thus, in specific contexts that value low power solutions,

such as edge computing for Internet of Things (IoT) devices, FPGAs provide an interesting balance between different hardware characteristics, combining better energy efficiency, customizable hardware that results in high execution speeds, and other advantages. As such, they are good hardware choices for hardware accelerators for artificial neural networks, but the necessary hardware design expertise and long development time limit this use of FPGAs (MITTAL, 2020).

Consequently, there is interest in overcoming the design cycle difficulties of programming artificial intelligence hardware accelerators for FPGAs. A possible approach is the creation of an automatic hardware description code generation system. This would compartmentalize the hardware design expertise, and it such knowledge would no longer be necessary for teams that wish to implement a hardware accelerator for their machine learning applications.

The automatic generation system would receive a trained neural network as input and output code to synthesize a hardware accelerator that implements the trained neural network.

The final system consists in a central interpreter core that would interpret whichever network loaded, and with the help of a function library, write the necessary code. As such, the main importance of this work is the generator and interpreter core that aims to be as generic as possible and be capable of producing accelerators for different networks, as well as the resulting development and expansion methodology. Limitations concerning the breadth of compatible neural network algorithms can be overcome by implementing them and adding to the central generator and interpreter. Consequently, the results obtained by generating and testing the hardware accelerators, in addition to testing their functionality, also demonstrate the main core's viability and potential to be expanded to include more functions to tackle different and more complex neural network problems.

1.1 Objective

This research aims to advance the area of automatic hardware description code generation for artificial neural network hardware accelerators.

Another goal is to define requirements for such a system to be successful, via the study of previous works and surveys of the existing field of research.

The existing gaps in the current state of the art of this area are studied and evaluated in this work, and research concludes that there are areas of possible improvements and that the current solutions do not satisfy the previously defined requisites. Thus, the final objective is to propose a novel architecture and workflow, which will be tested via the implementation of a functional proof of concept that meets the requirements, and whose presentation and further development brings significant advances to the area.

1.2 Motivation

There are already well established and successful neural network compilers and compilation flows for CPU and GPU target hardware, in general included with machine learning libraries that also function to create, train and test the models.

However, the area of interpreting and automatically mapping neural networks to hardware accelerators targeted at FPGAs is in its initial phases, with various approaches with different strategies with their respectives pros and cons.

In the context of devices for the Internet of Things, many demands arise from the restrictive nature of such an environment. Embedded systems on small physical objects that intermingle with day-to-day life bring significant problems as to their implementation. More specifically, considering the nature of edge computing, where intense computing is made in loco, such restrictions are fundamental to the design choices of systems to be implemented. Restrictions such as limited power availability, for example, a small device reliant on energy harvesting from a small solar panel, mandates that the computing on such a device must be especially efficient. FPGAs strike an interesting balance of providing customizable hardware which offers high execution performance paired with better energy efficiency than CPUs and GPUs, as well as smaller circuit area. As a consequence, FPGAs as accelerators for edge computing are an appropriate and interesting hardware choice (PLAGWITZ et al., 2021; MITTAL, 2020; ELNAWAWY et al., 2019).

Surveys of the current state of the area of automated neural network accelerator design flows conclude that this area is not yet mature, a lot of work is still needed and there is interest in further research and development of the field (PLAGWITZ et al., 2021; MITTAL, 2020). Proposed solutions that feature open source code of the same model of tools as the GNU Compiler Collection (GCC) are evaluated as of particular interest(PLAGWITZ et al., 2021). Another conclusion is that apart from the functioning and usefulness of the resulting hardware accelerators, the generator systems should have high flexibility, and coverage of different neural network techniques, to ensure good system usability (PLAGWITZ et al., 2021). Furthermore, it is crucial that the resulting tool is easy to use (PLAGWITZ et al., 2021; MITTAL, 2020).

Even though many proposed systems and solutions of the field of research fulfill part of the qualities desired in a complete automated neural network hardware design, there is a significant gap between existing systems and one that promises to achieve all necessary characteristics. Additionally, it is essential that the proposal is scalable for the continuation of the system and its enduring usability and usefulness; the machine learning and neural network fields are vast and ever-evolving, and as such a system that aims to accompany their developments must be flexible and updatable.

Many solutions under development of the area consist in proprietary projects, many times with code and resulting knowledge unaccessible to the public (KATHAIL, 2020). The current model of open source development methodology is evaluated as having high compatibility with the volatility, large scope and rapid evolution of machine learning techniques. Furthermore, the interest and demand of each specific neural network or machine learning technology orients the development and its inclusion in an open source hardware accelerator generator system. Surveys of the current state of the art (PLAGWITZ et al., 2021) evaluate that proprietary, closed source systems are significant obstacles to the flexibility of the resulting system, as it does not offer possibilities of natural, interest-guided extension of their libraries and resulting coverage through open research by the community.

As such, we conclude that the proposal of a system whose architecture fulfills simultaneously all of the desired qualities of performance, flexibility, open source code, ease of usage, and scalable, upgradable and modular design.

1.3 Methodology

Firstly, through reviewing the bibliography of already existing approaches in the field of study of automatic systems for the design of neural network accelerators for FPGAs, system requirements must be formally defined. These requirements must be defined in a way that a resulting system that satisfies them all corresponds to a functional, useful framework that fills the gap diagnosed in the area of study. Moreover, the requirements must be written as being easily testable and its characteristics directly comparable to other existing systems.

The first category of requirements pertain as to the resulting code being open, legible and well documented to promote the project's longevity and scalability. Then, the next category ensures that the system must be modular, flexible and must promote ease of compatibility by design. As such, its expansion and use in different neural network techniques may be facilitated, and increasing the project's durability. Additionally, the resulting system must be easy to use, since its central purpose is to facilitate the accessibility of hardware accelerator programming to a wider public which detains less specialized knowledge. Finally, the system must generate useful hardware accelerators that must be tested and compared as to significant metrics that are important to the context of edge computing. These metrics are FPGA size necessary for synthesis, execution speed and latency, and energy efficiency.

The objective of this research will be considered to be accomplished once a system architecture is proposed that satisfies simultaneously every requisite by design, and as such, bridging the gap in the area. Furthermore, a minimal and functional proof of concept of the system is also necessary beyond the system's specification to demonstrate its viability and future expansion into a more usable tool.

The hardware accelerator generator system was developed using the Python 3 programming language. The generator system imports trained neural networks and generates VHDL code that can be synthesized to implement a hardware accelerator for that specific neural network. This VHDL code is generated using functional block libraries written in VHDL, tested and simulated using the Modelsim software. Then, the generated accelerator was tested using the Intel Quartus Prime software, which provides several reports about the viability of the hardware's synthesis on FPGAs and

performance metrics such as execution speed and energy expenditure.

The resulting system was published using the GitHub (GITHUB, 2020) internet hosting system (MUYAL, 2022), under the GNU General Public License v3.0, which can be downloaded and used by executing the main function under the qNPUInterpreter file and importing the desired exported trained neural network file to be accelerated. Once the generator is finished, it outputs the necessary VHDL files to synthesize a hardware accelerator, which can be imported into an electronic design automation software such as Intel Quartus Prime or Xilinx's Vivado to synthesize the design into an FPGA.

The Github repository has instructions that detail how to expand the system, which can be done in 2 different diretions. First, to include new activation functions or neural network topologies, by developing new functional blocks in VHDL and changing the base Python interpreter to acknowledge the new additions to the library. Lastly, adding more compatible neural network development frameworks, which can be done by including converters from the framework in question to the Open Neural Network Exchange (ONNX) format to the Python Interpreter, and changing the import function to expect the new format.

To generate the test results, artificial neural networks were programmed and trained in the Python 3 language using the Tensorflow machine learning library, with Keras as an interface library. Additionally, the final system uses the Tensorflow Lite quantized neural networks. The networks used are more thoroughly described in the 8, but they were chosen in order to offer clarity in the results, use and test all of the implemented functions and topologies, and study the scalability of the synthesized final hardware.

The networks, once trained, are exported using the model export Keras functions, which generates a .tflite file. To generate a hardware accelerator, run the qNPUInterpreter.py file, use the graphical user interface to select the exported .tflite file to be imported, and wait for the software to execute. Once done, the system outputs the .vhd files which can be imported and synthesized to implement the accelerator.

1.4 Summary

This document is divided in eleven chapters. Chapter 1, consists in the research introduction.

Chapter 2, Literature Review, presents the context and theoretical foundations upon which this research is based on. Furthermore, the chapter presents a study of the automatic neural network hardware accelerator synthesis workflows field of study's current state of the art, reviewing multiple different projects, and drawing conclusions as to gaps in the field. Bridging these gaps consist in this project's motivation.

Chapter 3, Proposal, consists in a review of the context in which this project is inserted, as well as the scientific question to be answered. From this emerges the main objectives of the research, as well as architectural objectives and choices to fulfill them. Furthermore, this chapter describes the considerations over the source code's publishing model and explains different software licenses, as well as the final license choice made and its reasons.

Chapter 4, Prototype and intermediary results, presents a prototype made to guide the final version of the proof of concept. In this chapter, the prototype's objectives are explained, the methodology and work done is shown, and the results obtained as well as a discussion and conclusions drawn from them are presented. This prototype served as a base for the final version, and served to explore the field of work as well as design choices made.

Chapter 5, Accelerator Algorithms, begins the description of the final system by explaining all the important algorithms used in the project. This chapter begins by describing the variable quantization algorithm, which is not directly implemented in the hardware, but explains the variable representation formats used by the accelerator. These algorithms also include those implemented directly in hardware, namely the algorithms corresponding to the neural network layer topology, the neuron activation function, and the layer quantization corrections.

Chapter 6, Generator System, explains the generator system's current architecture. Furthermore, the chapter explains how the system is to be expanded in the future, including design guidelines for the different modules.

Chapter 7, Accelerator Hardware Architecture, shows the generated neural network hardware accelerator architecture, explaining the hardware functional blocks pertaining to neural network topologies, neuron activation functions, quantization corrections and the top hierarchical level entity that organizes the blocks.

Chapter 8, Tests, explains the methodology used to conduct the tests to validate the system.

Chapter 9, Main results, shows the generated hardware accelerator practical results obtained during simulations and compilation, as well as comparisons.

Chapter 10, Conclusions, reviews the entire project and comments over the objectives determined for this research, and whether the results obtained and presented successfully fulfill those objectives. In addition, this chapter analyses potential threats to the validity of the results, mentioning potential flaws of the testing process and the current state of the system and identifies limitations of the conclusions that can be drawn from these tests.

Finally, Chapter 11, Further work, takes reasoning from the Conclusions and Threats to validity chapters, and propose potential avenues for further work in order to expand and improve the project's system and conduct more and more significant tests over the existing system.

2 LITERATURE REVIEW

2.1 Context

2.1.1 Artificial Intelligence and Machine Learning

There are many different approaches to define Artificial Intelligence (AI). In the book Russell e Norvig (2021), the authors present and discuss different ways to reach a formal definition, with several approaches relating artificial systems and comparing them to human thought processes. Whichever definition is better suited to formally describe Artificial Intelligence and understand its meaning, the answer lies within the philosophy of artificial intelligence, and does not particularly concern the scope of this research.

What does concern this research, however, are the many possible and interesting applications of artificial intelligence techniques such as the field of machine learning. Machine learning has been defined as the field of study of artificial intelligence algorithms that learn and improve without explicitly being programmed to do so. They generally improve with experience and data, extract patterns and infer conclusions. (MITCHELL, 1997)

Machine learning has several different applications in a vast array of different areas, with useful, practical results and research has shown great promise. Examples for areas of applications of machine learning are robotics (MA et al., 2020; BOJARSKI et al., 2017; ENDSLEY, 2017), natural language processing and translation (WU et al., 2016), recommendation systems (DAS; SAHOO; DATTA, 2017), computer vision (RUSSAKOVSKY et al., 2015; HE et al., 2015), medicine (PICCIALLI et al., 2021; DING et al., 2019), among others (RUSSELL; NORVIG, 2021).

Machine learning results in several areas show promise and already have good practical implementations and results. Metrics such as accuracy, processing speed, pattern recognition and cost are shown to have good results, many times surpassing those of humans performing the tasks being automated. As such, there is considerable interest in expanding and developing these techniques, to better their results and

implement them in other settings.

2.1.2 Artificial Neural Networks

Of noteworthy mention among machine learning techniques are the artificial neural networks (ANNs). ANNs are computing systems inspired by the topology of biological brains, consisting of its basic unit, neurons, interconnected and organized in networks. Each artificial neuron receives signals, executes a determinate operation, called an activation function, and outputs a signal related to the result of this operation. The neuron's inputs can come from the output of other neurons, the network's own inputs, or be fixed values, such as a trained bias. Similarly, the neuron's outputs can go to other neurons, or become the entire network's final outputs. Furthermore, each neuron signal connection is usually multiplied by a weight, which is a numerical value obtained during the neural network training. All of these parameters, including the network's topology, number of neurons, their topology and interconnectivity, weights and biases are factors that the designers can control to fit each demand and problem to be solved. (RUSSELL; NORVIG, 2021)

Artificial neural networks are typically organized into ordered layers of similar neurons that pass signals from the first layer to the last. The first layer is the input layer, which receives the whole network's inputs, and the last layer is the output layer, whose results are the network's final outputs. The network may have other layers between the input and output, called hidden layers. Usually, neurons within a single layer execute the same activation function, and are interconnected to neurons of other layers following the same rules. There is a great variety of different types of activation functions, and the more commonly used are the following (RUSSELL; NORVIG, 2021):

1. The logistic or **sigmoide** function:

$$\sigma(x) = 1/(1 + e^{-x})$$

2. The **ReLU**, or Rectified Linear Unit:

$$ReLU(x) = max(0, x)$$

3. The **softplus** function:

$$softplus(x) = log(1 + e^x)$$

4. The **tanh**, hyperbolic tangent:

$$tanh(x) = \frac{e^{2x} - 1}{e^{2x} + 1}$$

Beyond the great variety in activation functions, there is also a diversity of different topologies. For example, the simplest is the densely or fully-connected layer. This is a layer whose individual neurons are connected to every single neuron in the previous layer. However, there are many other topologies that are used in different applications; one such example are convolutional layers, whose connections execute a convolution over the database or outputs of the previous layer. For example, in image recognition, a convolution can represent neurons that are specialized in specific parts of the image. All of the aforementioned topologies are feedforward, meaning the processing flows monotonically from the input layer to the output. However, there are also topologies with loops, called recurrent neural networks. (RUSSELL; NORVIG, 2021)

Recently, ANNs have become significantly popular and successful in a variety of different metrics, such as precision, processing speed, latency, data volume and scalability. For example, every machine learning work cited in the previous section utilizes some form of neural network. Notable ANN advantages come from the ease of use of resulting trained models, besides the high result accuracy even when dealing with complex systems and large inputs. Additionally, the generation of models through many different and effective algorithms, which train models adaptively through self learning and actualization enable this technology's use in a wide array of areas with great potential and results. Furthermore, ANNs exhibit great potential of high speed processing with its massively paralelized architectures (IZEBOUDJEN; LARBES; FARAH, 2014; ABIODUN et al., 2018). Rarely machine learning implementations use ANNs in complete isolation, and they usually interact with many different subareas of artificial intelligence and general software development, such as data science, statistics and probability. Moreover, they also include different workflow tasks beyond training and deploying ANNs, such as organizing and collecting relevant data or executing other kinds of processing. However, ANNs still are the central processing algorithm of these applications.

Artificial neural networks, when analysed through a black-box model, consists in a system which receives data to be analysed by the AI as inputs, and its outputs are results obtained through this analysis. The inputs can be extremely varied, and may be conveniently chosen depending on the subject matter and information deemed important for the analysis. Examples of possible inputs are images, possibly represented by their pixel numerical information, which can be medical exams or camera feeds, or other types of sensor data, such as temperature or pressure. Possible outputs, for example, can be classification results or regressions. In a classification neural network, the AI classifies each input sample into a determinate class, such as classifying whether an image shows a cat or a dog. In a regression, the network outputs a numerical result, such as a probability or a numerical value prediction. (RUSSELL; NORVIG, 2021)

The first step in implementing a neural network consists in its design, where several choices as to the architecture, layers, neurons and activation functions are made. Once designed and before the network is implemented to do actual tasks, it must be trained. Training consists in the use of an algorithm, which is executed over the network, typically using a database relevant to the problem to be solved by the AI. This database is made of possible inputs the system will eventually be exposed to, denominated a "training dataset". The training algorithm is iterative, and in every iteration the network is subjected to the inputs, generates outputs, the algorithm calculates an error function based on the correctness of the answers given, and modifies the network with the objective of reducing these errors. Over many iterations the internal network parameters are modified as to minimize the error function. Once the network's measured accuracy is deemed sufficient, the training stops and the parameters are stored, and the network with these trained parameters may be implemented in real applications.

It is worth noting that there are certain kinds of named layers such as pooling layers. These are effectively convolution layers that act on the previous layer's outputs executing activation functions such as max-pooling, that obtain the highest numerical value in each convolution window. This can be implemented through normal convolutional layers with activation functions that calculate maximum, without the use of weights or bias in its inputs.

In conclusion, it is clear that there is a vast range of different technologies and tools

in the field of artificial neural networks, each with its own application domain, theory and potential.

The basic neural network theory has its beginnings in the early 1940's, but recently it has experienced a significant resurgence of importance and practical use. This is due in part to recent advances in hardware technology, which results in cheaper and more accessible computational resources to execute training and application of neural networks, enabling its wider usage (PLAGWITZ et al., 2021). The training and implementation of artificial neural networks strongly depend on the hardware devices on which these processes are executed. As the neural networks and the problems they solve become increasingly larger, more important and more complex, their computational demands have increased significantly (MITTAL, 2020). As a consequence, there is great interest in studying the specific hardware to implement neural networks, including the creation of specialized hardware to accelerate such applications.

2.1.3 Hardware accelerators

A hardware accelerator is a specialized hardware that executes specific software more efficiently in comparison with its code being executed in a general purpose central processing unit (CPU). Hardware accelerators are usually employed alongside a CPU as a peripheral, executing only their specialized task for the system, while a CPU executes the rest of the more general tasks.

An example of such a system in the context of the Internet of Things could be an embedded system deployed in a restrictive environment, such a small smart camera. The device, among other components such as batteries or antennas, could consist in the camera itself, providing input data to be processed, a CPU that executes and processes data in general, and a hardware accelerator connected by hardware to the CPU as a peripheral to execute a neural network over the collected data, and output the results back to the CPU.

As such, hardware accelerators are physically connected to the CPU and are a subcomponent of the larger Internet of Things system, and are otherwise isolated. Consequently, concerns over data breaches or security of the Internet of Things are issues to be handled by the design of the rest of the system, such as internet data

protocols used during communication between devices, not the underlying design of the hardware subcomponents. This peripheral is subject to a physical connection with a main/peripheral asymmetric control with the CPU, and apart from a physical breach and connection to the FPGA electronics, the data between the two components is secure.

Many different metrics can be used to evaluate what consists in a performance improvement, such as processing speed, energy efficiency and integrated circuit area used.

An important design choice to be made is the hardware on which to implement a hardware accelerator. Historically, artificial neural networks were executed mainly in CPUs and Graphics Processing Units (GPUs), but other possible options are Field-Programmable Gate Arrays (FPGAs) and Application-Specific Integrated Circuits (ASICs). The increasingly extensive use of GPUs for purposes other than graphics processing, such as artificial intelligence, has coined the term General Purpose Graphics Processing Unit (GPGPU)

FPGAs have several attractive features for designing hardware accelerators for artificial neural networks. These devices can be configured after their manufacturing by a designer to implement hardware specialized in a specific algorithm, allowing its execution with a higher energy efficiency than GPGPUs and CPUs, and higher execution speed than CPUs (MITTAL; VETTER, 2014) .

Another important characteristic is that FPGAs can be reconfigured indefinitely by the end user after the circuit's production. As a consequence, these devices are a good fit for hardware accelerators for machine learning, as this allows prototyping of different implementations and parameters, adapting the hardware in the field. This is especially important for this area of study, as there is a wide array of different parameters and configurations of machine learning applications, its implementation is often iterative in nature, and allows the possibility of updating models in the field after their implementation. Furthermore, the field of machine learning technologies advances quickly, with new and practical architectures being constantly released and improved. While ASICs also provide significant hardware specialization, their long, complex and costly design cycle along with the fixed nature of its final hardware hamper its use as a more flexible

hardware acceleration platform for AI (MITTAL, 2020).

Many libraries and frameworks are available for development of machine learning solutions, and these often include tools to compile and implement the resulting software to be executed in CPUs and GPGPUs (ABADI et al., 2015; JIA et al., 2014). However, the design of hardware for FPGAs is still difficult and demands specialized knowledge of hardware design from the developers, and machine learning programmers not necessarily have it. Additionally, the area of workflows and frameworks for automated synthesis of AI accelerators for FPGAs is still in its early stages and much work is still needed (PLAGWITZ et al., 2021) This combination hampers the use of hardware accelerators for typical machine learning programmers, keeping away the advantages of its use, whose difficulties demand specialized designers.

As such, there is an interesting gap to be filled in an partial or total automation of synthesis of FPGA-based acceleration of machine learning algorithms. Such a system would bring hardware accelerators to wider audiences, facilitating its use without specialized hardware design knowledge. Furthermore, even more experienced designers can benefit from the use of such a tool, speeding up their work and helping workflows.

In Plagwitz et al. (2021), a wide-reaching survey is made in the scope of automated design flows and compilation of artificial neural networks in FPGAs. Several different projects are studied and tested, and its fundamental paradigms were classified in two different groups: dedicated circuit designs and overlay-based co-designs. Dedicated circuit designs correspond to the direct implementation of the algorithms used, mirroring their architecture and topology directly in hardware. On the other hand, overlay-based co-designs represent the implementation of common processing cores, specialized in the execution of operations used in the computation of the neural networks. Then, the neural network behavior is translated into instructions for execution in these specialized cores. These two different approaches are compared through testing of several projects of both categories, measuring metrics such as energy efficiency of the resulting systems, flexibility and compatibility, and comments on whether the projects are open source or proprietary. The work then concludes that the available solutions are insufficiently mature and do not address completely the existing demand for such systems. Fundamental problems in the existing solutions lie in the lack of system flexibility and compatibility with the diversity of neural network algorithms currently

in use. Additionally, it is diagnosed that the available tools are hard to use, without firndly interfaces to the end user. Finally, the work acknowledges that in general, most of the more advanced systems are closed source and of proprietary development. As a consequence, even is these tools are more flexible, they may not be expanded by the community and its use is often restricted to devices sold by the respective company. In addition, the work concludes that there is a significant lack of solutions for the scope of accelerators for edge computing. The only system found and studied was the Lattice Semiconductor's sensAI platform, which is also low flexibility, proprietary and closed source. In conclusion, an open source solution is deemed to be more fitting for the development of a more flexible and extensive automated workflow for the development of FPGA hardware accelerators. The development by an active community would keep up with the fast advancements in machine learning and neural network technologies, and high compatibility with different algorithms and model representations is fundamental.

2.1.4 Neural network quantization

When considering the context of implementing artificial neural networks in Internet of Things devices, there is a clear conflict between the high computational resource demands from machine learning algorithms and heavy power and size limitations derived from the environment's restrictions. As a consequence, many different approaches were created to balance these factors satisfactorily.

The first type of approach is to start the optimizations in the neural network's model design, exemplified by models such as the tiny version of the YOLO algorithm (REDMON; FARHADI, 2018), the Squeezenet architecture (IANDOLA et al., 2016), or MobileNets (HOWARD et al., 2017). These approaches are novel architectures that optimize their size and efficiency before its implementation by choosing efficient operations.

The other category of approach is to reduce model size and operation complexity through the reduction of bit-depth representations of system variables. Applications such as Binarized Neural Networks (HUBARA et al., 2016), XNOR-Net (RASTEGARI et al., 2016) and Ternary Weight Networks (LI; ZHANG; LIU, 2016) all use significantly reduced precision while preserving accuracy in complex applications. Within the cate-

gory of lower bit-depth representations, the TensorFlow Lite's Integer-Arithmetic-Only quantization (JACOB et al., 2017) stands out. TensorFlow Lite's solution, differently from the other's, was tested beyond model size reduction, and arithmetic using the lower representation sized also optimizes execution speed in real, mobile hardware, while preserving model accuracy to nearly-identical levels to the initial model.

Within the category of model compression by reduction of bit-depth representations, there are several advantages to be considered. One such advantage is that any original neural network that internally uses floating point parameters may be quantized, preserving its overall architecture. As such, the model doesn't need to be designed from the beginning considering its compression. Furthermore, the level of compression can be scaled according to the demands of the application using different bit-depths. While the Binarized Neural Networks only use binary weights (HUBARA et al., 2016) and are significantly compressed, TensorFlow Lite's quantization use 8 bit and 16 bit integer and fixed point precision (JACOB et al., 2017), which demonstrate significant model accuracy preservation.

Another interesting consideration is that arithmetic computed using lower bit-depth representations is also faster and more efficient, as the operations are simpler and require less computations. Furthermore, if implemented in specialized hardware, this also translates into smaller dedicated circuit area to implement these simpler operations.

In conclusion, for the implementation of a hardware accelerator in a restrictive environment such as Internet of Things edge computing, analyzing model complexity and its representation bit depth should be considered in the trade-off between hardware use and application performance. Hence, a more complex and precise model, when quantized, can fit into a smaller FPGA with better results than a simpler model using unnecessarily high bit precision. Furthermore, quantization can speed up neural network inference speed and reduce energy costs in hardware by simplifying the operations necessary for computation.

2.2 Related works

By researching recently published scientific articles, proprietary solutions from competitive companies from the area, and promising projects cited in meta researches such as Plagwitz et al. (2021), many examples of FPGA neural network hardware accelerator compilation and automatic synthesis workflows were brought up and analysed.

One such example is the Vitis unified platform (KATHAIL, 2020), the Xilinx commercial solution for a development workflow of artificial intelligence acceleration for FPGAs sold by the company. It includes a closed source code compiler, and consists in an overlay based co-design approach. Kernels are synthesized in the FPGA according to the device's size, and the neural networks are converted to instructions for these kernels. A pre-trained, compatible network library is divulged, however, all of the nets are convolutional. The platform is compatible with model representations in Tensorflow, Caffe and Pytorch. Furthermore, the processor implements network approximations, quantized to 8-bit representations. However, the co-design architecture has its downsides, the synthesis is limited to proprietary hardware, and the code is unmodifiable and inaccessible.

The sensAI tool (LATTICE. . . ,), from the Lattice Semiconductor company, is a commercial product developed for neural network accelerators focused on FPGAs with low energy consumption, such as the iCE family of devices, sold by the same company. This is also a closed source, proprietary solution, and, similarly, the synthesis targets are exclusively the company's FPGAs. Beyond that, the compatible function library and available network topologies is relatively restricted, with support for only convolutional nets, and restricted accessibility to the code prohibit the libraries' expansion by the community. The compatible representation formats are Tensorflow and Caffe. The overlay-based co-design methodology was also chosen, as there is control over what hardware is used, given that the tool is only compatible with proprietary hardware; as such, kernels can be developed targeting these products. Furthermore, small devices with high energy restrictions have high limitations and designing a small kernel that fits the FPGA can enable bigger networks to be synthesized and executed. However, we conclude that the limiting and closed source function library are significant gaps in this

solution.

The OpenVINO (INTEL, 2021) tool, from Intel, stands out as, even though it is a commercial solution, makes a version of its code open to the public, through the use of the Apache 2.0 license. However, it is just a part of the total project, and the target devices are limited to Intel hardware. One other interesting characteristic is that FPGAs are not the main focus, which has been discontinued recently, and instead favouring heterogeneous execution between CPUs, GPUs, and proprietary specialized hardware called Neural Compute Sticks, as long as they are Intel's. Nevertheless, the company published a series of auxiliary softwares that aid neural network design and optimization, accelerator development, and offers compatibility to a wide variety of algorithms and topologies. Furthermore, many network model representations are compatible, such as Tensorflow, Caffe, ONNX and MXNet, along with a large layer library and pretrained networks. This approach also addresses network and data quantization by featuring a quantization toolkit which supports arbitrary bit depth precision. This toolkit was tested and 32 bit models quantized to 8 bits showed better execution performance (GORBACHEV et al., 2019). In conclusion, this framework is useful for neural network inference in heterogenous hardware systems employing various Intel devices. The diverse function library published and its compatibility with several representation formats are clear strengths, possibly correlated with structured development by the corporation, but also with the fact that the code is open source.

Another overlay-based implementation is the Versatile Tensor Accelerator (VTA) (MOREAU et al., 2019), that is part of the TVM open source code compiler. As its name indicates, the tool implements processing kernels that accelerate tensor operations, useful for neural network execution. However, originally specified for use as a peripheral to process part of the computation as part of a heterogenous processing system, alongside with CPUs and GPUs, there is little support and friendly interface for the end user in relation to its use targeting exclusive use in FPGAs. Furthermore, its flexibility is relatively low, being only compatible with residual neural networks. In conclusion, while this project is architecturally promising, being developed with an open source code model, its flexibility and compatibility characteristics do not correspond to the demands in the field. Even though, it is an interesting project, whose modular design is noteworthy, which would facilitate its further expansion.

The DeepBurning system (WANG et al., 2016), also open source, is a recent and more academic project based in specialized custom circuit design. Many types of layers used in machine learning applications were deconstructed in functional blocks, which had their implementation described in hardware, and through a model's compilation they are coordinated into a neural network implementation. Additionally, an interesting strategy was utilized to overcome the circuit area and device usage restrictions on the target FPGAs: the use of "spacial folds" and "temporal folds". These techniques consist in the reuse of functional blocks during runtime by multiple different neurons. In spatial folds, neurons in the same layer reutilize the functional blocks, effectively "folding" the layer size, and in temporal folds, different layers reutilize hardware, folding in different execution times. This creates a stack of data to be executed sequentially, but the circuit hardware specialization is kept and there is still gains in performance. However, the code's publishing is made through many different incomplete repositories, whose last update was made in 2019, with empty folders, few documentation and instructions for the reproduction of the results or the system's expansion. Hence, while the system is relatively flexible and has interesting strategies to overcome the contradiction between high hardware size restrictions and the scale of computation of neural networks, its publication and project continuity efforts can be significantly improved.

Another interesting academic project is the Intellino processor (YOON et al., 2020), also based in specialized custom circuit design and synthesis in FPGAs. This project corresponds in the acceleration of two specific neural network algorithms: k-NN and RBF-NN. Although their implementation is efficient and has good results, shown in their article, it is a hard wired application whose alteration to other different algorithms would be hard. The reason is that the system is itself the blocks that implement the algorithms, without a reusable common core, and as such, other algorithms would have to be implemented from zero. The code's publishing, while open source, with python scripts for the libraries' installation, is scarcely documented, and the materials available in the website do not completely correspond with the algorithms presented. From this project, we conclude that their clear presentation of results and performance tests is interesting and important to properly evaluate and compare such systems. This work uses classic classification problems and datasets, recognized by pattern recognition and neural network academia, and as such, their results are easily recognizable

and comparable to others in the area. However, it is worthy of note that the hard-wired architecture design hampers the expansion of the system, and its modularization could facilitate its expansion and longevity. Furthermore, legibility, clear publishing and proper software license choices are similarly important.

In conclusion, through this analysis of related works, its possible to diagnose the gaps in the state of the art of this field of study. One of the main results obtained is that the open source development paradigm fits particularly well with workflows for automatic accelerator synthesis. Furthermore, the different projects studied were divided in two distinct categories concerning their main architectural choices, each with its own advantages and disadvantages: dedicated circuit designs and overlay-based co-designs. Finally, proper publication, legibility and ease of use are fundamental for ample adoption of the system and its further development by the community.

3 PROPOSAL

3.1 Context

An important design choice lies in the type of hardware utilized for the implementation of a accelerator. In the field of neural network hardware accelerators for embedded systems, the options are Application-Specific Integrated Circuits (ASICs), General Purpose Graphics Processing Units (GPGPUs) and Field-Programmable Gate Arrays (FPGAs).

ASICs circuits made specifically for a determinate use, and as such its circuit area and energy usage overhead is kept to a minimum with the employment of specialized, fully customized hardware. While this results in high performance and efficiency, there are several problems associated with its design cycle. This design method has considerably low flexibility, since the hardware's configuration is fixed and defined in its fabrication. As a consequence, if the design aims to be more generic to be compatible with multiple machine learning algorithms, its advantages diminish proportionally. Conversely, when fully specialized and optimized, the resulting hardware only executes a single algorithm. As such, this hardware choice does not address the demand for a nonspecific platform for neural network acceleration. The rapid research and development of new neural network algorithms, network topologies and activation functions means that the field is in constant change; hence, the use of ASICs as the hardware choice for a common platform for a variety of hardware accelerators, each targeted to its specific algorithm, would cause significant efficiency and performance loss. This is due to the distance and incompatibility between the specialized circuits of each final hardware and the specific application being accelerated. Additionally, ASICs are burdened with hard and costly development, and the high specialization of the resulting devices cause an expensive and time consuming design cycle with low prototyping flexibility. (ESMAEILZADEH et al., 2012; WANG; LI; LI, 2016)

GPGPUs are another hardware option for the platform's implementation. This type of device also offers beneficial characteristics in relation to the neural network inference performance. Present-day machine learning algorithms exhibit intrinsically par-

allel execution calculations, with many independent operations which can be executed simultaneously for significant processing acceleration. GPGPUs, meanwhile, were architecturally derived from Graphics Processing Units (GPUs), which were specifically designed to efficiently calculate the similarly parallel operations necessary to render computer graphics. This structural advantage can be exploited by configuring GPUs with code to execute neural network operations. However, GPGPUs have a significant energy and circuit area overhead, which is particularly harmful for the context of implementing lightweight machine learning solutions for edge computing in IoT devices. The hardware choice for such an accelerator must account for the demanding energy and space restrictions which are intrinsic to this environment. (PLAGWITZ et al., 2021; ESMAEILZADEH et al., 2012)

FPGAs, meanwhile, offer an interesting balance between several features of other solutions, summed with its own unique structural advantages. This methodology provides more beneficial design cycles, which include ease of prototyping, updating and modifying hardware due to the possibility of reconfiguring its layouts after the device's manufacturing. Additionally, FPGAs also offer the possibility of programming truly parallelized computation in its architecture in the implementation of a neural network algorithm, resulting in significant performance and energy cost benefits. However, once manufactured, each FPGA device has a fixed circuit area with a ser of internal components, that, if not used in the final design, result in a circuit area overhead. Consequently, while FPGAs don't exhibit the very best performance in any of the aforementioned metrics, they do provide an attractive balance between noteworthy acceleration, relatively low energy cost and area use, while being significantly more flexible, cheaper and easier to design projects that target them in comparison with the other hardware choices. FPGAs are also widely used in successful machine learning applications.

However, the design of hardware acceleration solutions for synthesis in FPGAs have high design complexity, and most importantly, demand from the developers specialized hardware description language (HDL) technical knowledge. HDL programming is a relatively uncommon proficiency, and machine learning programmers not necessarily have it, reducing the availability of hardware acceleration for the majority of such development teams (MITTAL, 2020).

Hardware accelerators act as a peripheral subcomponent which is part of a larger

Internet of Things system, physically connected to a CPU and are otherwise isolated. An example of an Internet of Things network of devices could be a network of connected smart security cameras. These cameras could have circuit boards with CPUs that process general data and are connected amongst themselves via bluetooth antennas, for example. Each device's circuit board with the CPU would include an FPGA, physically connected to the CPU. The FPGA is completely isolated from the outside world, and would communicate with the CPU via hardware using a master/slave dynamic, with the CPU sending neural network processing data requests, and the FPGA sends the processed data results. Apart from a sophisticated physical internal electronic hardware interception security breach, Internet of Thing's security issues concern the system as a whole, communication between systems and data security. Such a hardware interception attack would consist in physically modifying the circuits to intercept electric communication signals between the CPU and the FPGA acting as hardware accelerator, reading and interpreting them, which would be unrealistically difficult, and if an attacker already has the kind of access needed to perform such an attack, there are already worse problems. As a hardware peripheral, the only interaction the accelerator has with the outside world is a physical circuit connection with the system's CPU, and is subject to a master/slave communication dynamic, being otherwise isolated, with no other inputs or vulnerabilities, such as connection to internet. Consequently, concerns over security are on the abstraction level of devices, and are outside of this work's scope.

3.2 Objectives

The literature review made in this work concludes that there is a significant gap in the current state of the art of neural network hardware accelerator generators targeted to edge computing devices. The first of which is lack of generality and flexibility of the current systems. Many of the available systems only accelerate a single or small group of similar algorithms, and are designed in such a way that they are difficult to adapt to accelerate other techniques. They consist in hard wired systems whose architecture targets specific types of algorithms, and as such significantly lack breadth of compatible applications. This markedly limits their implementation to applications and contexts different than their original targets, and hinders their expansion and longevity. Addition-

ally, the compatibility in relation to different model representations is a notable obstacle to the system's generalization. If the system fundamentally depends on specific model representations, its compatibility with other models is harder to implement. Another existing gap is the partial or complete unavailability of the system's source code, by a variety of reasons. This significantly complicates or makes impossible to reproduce the results obtained in such researches, to use the system as a tool in a project, or to work on its expansion. Furthermore, another problem diagnosed in the state of the art is the insuficient or inaccessible code documentation, additionally interfering with their system's use by others. Additionally, from preliminary testing of potential architectures and intermediary results we conclude that the inclusion of neural network model quantization brings significant advantages to the design. As such, another gap in current solutions and potential advantage with the proposal of this system is the inclusion of such quantization, which promises to increase the resulting system's inference speed, energy efficiency, reduce model size and preserve most of the original model's precision.

Consequently, the main objective of this work is the development of an architecture for a FPGA hardware accelerator generator which targets artificial neural network applications for the context of edge computing, with a proof of concept implementation that sufficiently addresses the gaps diagnosed in the current state of the art. In addition, this proof of concept must sacrifice the minimum possible of performance and efficiency as long as the resulting system still accomplishes its main purpose of accelerating algorithms in a synthesizeable, adequate way considering the restrictions imposed by the Internet of Things' environment.

3.3 Scientific question

To properly define the objective for this work, the following scientific question is raised:

Is it possible to create an edge computing neural network hardware acceleration generation system architecture that bridges the gaps in the current state of the art?

For the successful and acceptable satisfaction of this question, the creation of a proof of concept that fulfills every objective is necessary. As such, the system has the

following requisites:

1. The system must generate a successful hardware accelerator.

   This entails that when executed, an HDL code must be generated, that when synthesized has clear performance gains in the execution of an artificial neural network when compared with a similar CPU.

   Since the target environment, Internet of Things devices, presupposes restrictions in energy consumption and small device sizes, the final hardware must be measured in more metrics than solely execution speed in order to consider its success.

   As such, the hardware will be measured in regards to energy expenditure, circuit area and execution speed. These metrics will be compared to similar solutions, and an interesting balance between these results must be achieved for the accelerator to be considered successful.

2. The system must generate a hardware accelerator which is compatible with edge computing restrictions.

   The context of edge computing for embedded systems demands for low power, small applications. Even if the system is efficient, it must fit in FPGAs with small footprints and energy expenditure.

3. The system must be open source, with legible and well documented code.

   As such, reproducing research results and implementing different hardware accelerators by third parties to implement different algorithms and artificial neural network applications will be significantly facilitated.

4. The system must be modular by design.

   To facilitate the project's expansion by the community, its architecture must allow the independent programming and inclusion of modules which implement different algorithms, but continue to be compatible with one another. The system will then act as a hardware generation common core, used by the different modules, with design guidelines for the inclusion of other techniques.

5. The resulting tool must be easy to use by developers that do not have HDL hardware design expertise.

   As such, the interface with the end user must completely isolate the HDL knowledge, only requiring well-defined inputs and outputs, accessible to artificial intelligence software programmers.

3.4 System Architecture

The proposed system can be conceptually divided in two subsystems. The first subsystem is a VHDL code generator which imports and interprets neural network models to generate code. The second subsystem is the generated VHDL code itself.

Both subsystems must comply with the open source and ease of use by the end user requirements. Additionally, the code must be legible and well documented, and include a compatible open source code license. Furthermore, the end user must not be required to have specific knowledge of hardware design to use the final tool.

3.4.1 Subsystem 1: Generator

The generator subsystem consists in a software which receives artificial neural network models in compatible representations and algorithms used, interprets it and generates VHDL code. This code, when synthesized, implements a hardware accelerator that executes the neural network received as input.

To ensure ease of use, the system only requires the model as input. The subsequent interpretation and hardware generation is made completely automatically, using the data found in the model's interpretation.

Furthermore, this subsystem has as its only output the VHDL code ready to be synthesized, and the end user does not need to interact with HDL level programming.

The current system proposal is compatible with the Tensorflow neural network development framework, widely used by several applications and researches with great success (ABADI et al., 2015). However, the system is made in a way that input models are first converted to the Open Neural Network Exchange (ONNX) format (BAI et

al., 2019), which is an open source artificial intelligence format that defines a common set of operations for neural networks. As such, any format which is compatible with the ONNX exchange representation is also compatible with this work's core generator system, and the only change necessary is the inclusion of each specific framework's converter. Since ONNX is focused on interoperability, the systems compatibility includes several popular frameworks with readily available open source converters, such as Caffe, Keras, Matlab, PyTorch, SciKit Learn and many others.

The system's execution begins with the import of an artificial neural network. The network must already be trained. Then, the software extracts the network topologies and activation functions used. The network topology consists in the amount of network's inputs and outputs, number of layers and neurons in each layer, and connectivity pattern between neurons. The activation functions consist in the function executed internally to each neuron. Then, the software extracts the trained parameters from the network, such as weights and biases, and matches them with each layer in a specific class data structure.

Once all the necessary data is obtained from the model's interpretation, the generator subsystem accesses a functional block library which contains neural network VHDL hardware implementations. This library contains both activation functions and topology configurations used to generate the final VHDL code. This architectural decision is part of the effort to modularize the system's design. The consequence of this design decision is that each algorithm and topology is isolated and organized in a library, has clear design standards, and for the inclusion of new techniques a developer can implement a functional block following compatibility standards and include it in this library without editing the surrounding code. Furthermore, the modular and standardized design pattern facilitates the system's legibility and documentation, helping further development by multiple independent developers such as it is in an open source context. Another consequence of the modular system is ease of testing, as each component can be isolated and tested before its inclusion in a larger, more complex system.

Finally, the generator subsystem writes a VHDL program using the code found in the activation function and topology functional block library, and configures the blocks according to parameters extracted from the model's interpretation. The final code is ready to be synthesized.

3.4.2 Subsystem 2: Hardware Accelerator

The second subsystem is the final VHDL code generated by the generator system. The hardware accelerator must fulfill the system requirements pertaining to execution performance and energy efficiency, demonstrating a significant gain in the inference throughput and energy expenditure balance. Furthermore, the resulting hardware must be synthesizeable and fit in a suitably-sized FPGA, which in the edge computing for Internet of Things devices means a small, low-power device.

The subsystem's architecture can be divided in layers and network topology.

Firstly, the network is organized in layers of neurons, which in turn are the basic units in neural networks. Each layer has data inputs, which can come from previous layers or the network's own inputs, and has its data outputs, which can go to other layers or the network's final output. To write the code related to each layer, the generator subsystem uses a sample VHDL functional block of the same type of layer, and configures its parameters according to the network. The parameters to be configured are number of neurons in each layer, number of inputs, number of outputs, amount and value of weights and biases, variable data types and functional internal parameters. Then, the code corresponding to the activation function requested is obtained in the library and written in the layer.

Note that this manner of dedicated circuit design, with specific, specialized hardware that mirrors the neural network topology, corresponds to a design decision made to fulfill performance and energy efficiency requirements. As discussed in the literature review and context of hardware accelerators, dedicated circuit design optimizes execution speed and efficiency, allows truly parallelized computation of independent operations in neurons on each layer. Furthermore, hardware pipeline techniques are implemented for clock frequency speedup and increase in performance as each hardware component can be more fully utilized, minimizing waiting times for previous results. The pipelining techniques utilized in the layer topology of the neural network hardware consist in registers between each layer, which store the layer's output values. As a consequence, when one layer is done processing a request, it can begin processing data related to the next request instead of waiting for the entire system to finish. As such, with enough requests, every layer can be working simultaneously on different

requests, whose results are held in properly identified buffers. This results in significant performance optimization, with the hardware being more fully utilized. Meanwhile, the dedicated circuit design helps with the system's modularization, as each specific function consists in its own specific dedicated circuit that can be separately developed and tested. This aids the project's scalability and its open source organization, as different developers can independently follow design parameters and program different functions.

In addition, as discussed previously, the optimizations related to the quantization of the variables and parameters used in the neural network are also included as separate layers, adjacent to the corresponding quantized layer. This is done in order to also pipeline the calculations executed to properly utilize the data represented in the quantized format and more fully optimize the hardware. As such, after each functional block that uses quantized data types, there is another functional block which utilized the corresponding variable's, weight's and biases' quantization parameters to calculate correction factors to the layer's outputs. Note that this architecture also follows the dedicated circuit design approach, and is also fully modularized and present in the neural network functional block library.

The second part of the subsystem is the network's topology. The topology is represented by a VHDL component which configures the neuronal interconnectivity, mapping inputs and outputs according to the imported neural network model for its correct execution. Similarly to the layer's activation functions, the generator subsystem imports connectivity components from the functional block library. Once configured, the network topology component acts as the network's interface, receiving the input data to be processed by the neural network, receiving and emitting control signals, and outputting the final results obtained by the artificial intelligence.

This design is similar to the layer's architecture, as the topology's component also corresponds to a dedicated circuit design, fully exploiting FPGA's custom hardware synthesis to optimize execution speed and efficiency. The topology component also instantiates the registers that act as buffer between layers, enabling their pipelining. Additionally, the topology component is also modularized, as it draws from the functional block library and can be compatible with multiple possible topologies once they are included in the library.

3.5 Open source license

In this work, we have addressed many facets of the current state of the art of automated workflows for generation of hardware accelerators for neural networks.

In the literature review, we studied many different approaches to machine learning design automation flows. One of the main considerations made while analysing other projects was whether or not the source code was freely published under an open source license, or it was a proprietary, closed source project. While studying the existing solutions, there were many different perspectives used in this regard. There were both academic and corporate projects with the code fully published, and that accepted contributions from the open source community;

It is worthy of note that even though some solutions, such as Intel's OpenVINO (INTEL, 2021) is proprietary, the corporation judged that it was positive for the project and the company to divulge a version of the code under the Apache 2.0 license. Since the synthesis part of the software is closed and only targets Intel's proprietary hardware, the open source, machine learning part of the software draws attention and directs the public towards purchasing their hardware and other solutions. Another consideration is that the vast and ever-changing nature of the machine learning and neural network state of the art is compatible with the open source development paradigms, as it confers the library more flexibility and agility to keep up with new developments and technologies, powered by the demand of the implementation of such algorithms.

On the other hand, academic solutions such as the DeepBurning system (WANG et al., 2016), also open source, has the disclosure of its code as one of its main objectives, as is the publication of papers over their updates and achievements. However, while proprietary solutions often disclose only a subsection of their code, motivated by the profit from retaining the remaining intellectual property and control over the source code, academic solutions not particularly concerned with the continuity of their projects by the community often have many issues with the code publication. For example, publishing in different, incomplete repositories, with empty folders, poorly documented code and tools, and few instructions for the reproduction of the results cited in the papers or for the system's continuity.

From this analysis of the code development model, we concluded that the open source methodology shows a better fit with the field of study. Open source development allows for greater expansibility through research and development from the community; the project's development direction benefits from the guidance provided by the developers, as they naturally gravitate towards algorithms and technologies of interest due to their own demand; the field of study of machine learning and artificial neural networks changes rapidly and the system must be constantly updated and new technologies included for it to stay relevant, and open source development allows for this agility; and ultimately, the project's goal to provide an useful, open source system to divulge artificial neural network hardware can only be achieved by publishing the source code produced.

To this end, the entire source code of the project was published using a GitHub (GITHUB, 2020) repository. GitHub is an internet hosting service for software development whose use as an open source host for open source projects is popular. According to statistics provided by GitHub, over 94 million developers and over 4 million organizations use the service, and over 330 million repositories are hosted in this platform, and as such, is a good environment to host our open source system. Using such a popular platform will encourage contributions from the community and its tools, such as distributed version control and data integrity using Git will encourage the project's longevity.

Another consideration with the source code publication is the open source license chosen and used. Among popular choices are the Apache license 2.0, the MIT license, and the GNU General Public license v3.0. These three licenses are similar, with many of the same permissions, conditions and limitations provided, but they also have key differences. Firstly, all of these three licenses provide permissions for private and commercial use. This is deemed important for this project, even as it begins as an academic work with non-profit goals, because to be usable in commercial purposes and applications increases how useful the final tool is, and may bring more interest and another demographic of final users. Then, the licenses allow for distribution and modification of the source code, which is key to propagate the project make it available for the public, allow for contributions and for developers to adapt and implement the product as needed. All of these licenses also provide limitations of warranty and

liability, which is standard in such licenses. This is important to protect the developers of the system, as the code is provided "as is", and places responsibility on those that use and implement the tool at their own risk without punishing the project. As a consequence, by following standard practices in open source development and protecting the community, we encourage more contributions to be made.

However, these licenses differ in relation to the conditions under which the software may be used, modified and distributed.

The MIT license is the most permissive of the three, requiring simply the preservation of its own copyright and MIT license notices. As such, the MIT license propagates itself along the code that uses it, and guarantees that the source code remains open.

The Apache license, on the other hand, requires state changes to be documented by those who change the licensed material, as well as the Apache license and copyright notice preservation. The documentation of state changes is another protection of developers and authors, as problems of new, modified versions will not be erroneously attributed to the main versions, and also aids in the troubleshooting of those versions. Another permission granted by the Apache license is its express grant of patent rights from contributors. This entails that each contributor grants a perpetual, worldwide, non-exclusive, no-charge, royalty-free, irrevocable patent license, guaranteeing that the source code protected by this terms will remain free and will not be threatened by patents.

The GNU General Public License (GPL) v3.0 includes the conditions of the other two, being the documentation state changes to the licensed material and preservation of its license and copyright notices, the express grant of patent rights from the Apache license, as well as other interesting conditions. Differently from the other two, while the other licenses merely allow for distribution and modification of its source code, the GNU General Public License v3.0 has as one of its conditions the obligation to disclose the source code of the modified versions. As such, modified versions of the software are to be published under the same license, with the source code available, while the others allow for the distribution of closed source versions of the project. This means that the GNU GPU v3.0 is a strong "copyleft" license that is focused in sharing improvements and keeping the software and all its future versions free and open, while a project

protected by the other licenses may be superseded by a closed source version of itself, defeating the purpose of the code being open source.

In conclusion, by imposing conditions and rules to regulate the use, distribution and modification of the source code, the GNU General Public License v3.0 offers more guarantees of the continuity of the project and ensures that improvements to the system are kept open and free.

As such, the license chosen for this project is the GNU General Public License v3.0, and the source code has been published in a GitHub repository under this license. The limitations in warranty and liability provided by the license reasonably protect the authors and developers; the license is highly permissive, allowing for private and commercial use, provides a patent and allows the code's distribution and modification; and the conditions provided keep the source code free, preserves the license and copyright, and protects the versions of the code by obligating the documentation of state changes.

All of these characteristics adhere to the project's goals of being free, not hardware dependant, ensure the projects longevity and continuity, and being an useful tool for academic, private and commercial use.

4 PROTOTYPE AND INTERMEDIARY RESULTS

As part of the iterative development process, an intermediary proof of concept system was created to evaluate design choices and possible changes. This prototype was designed such a way to include all the steps necessary to a complete system, and as such, highlight potential weaknesses and difficulties to solve. The prototype's architecture is divided in four segments:

1. The design and training of a functional artificial neural network model, with measurable performance and which is compatible with the following steps;
2. A data extraction method which includes all the necessary information to reconstruct this neural netowrk model;
3. The creation of a prototype library, which includes hardware description language code corresponding to the implementation of the necessary activation functions and topologies used by the neural network;
4. The development of a system which uses the extracted data and library from previous steps and generates files corresponding to the neural network being interpreted.

Furthermore, it's necessary to create tests to validate correct function, to diagnose potential problems and evaluate performance.

4.1 Neural network design and training

To create a prototype which covers all of the fundamental phases of the project, it is necessary to design an artificial neural network to be used as an input which is compatible with the rest of the system. Since the system, in its initial form, will have a limited topology and activation function library, the neural network compatibility will be restricted. To represent the neural network field of study with as much brevity as possible, it was chosen to include the ReLU (Rectified Linear Unit) activation function and the fully connected, dense layer. These are among the most simple technologies

used presently, and still present good results in regards to accuracy and model complexity (RUSSELL; NORVIG, 2021). Consequently, the neural network designed to be used as example for the prototype must use only the ReLU function and dense, fully connected layers.

To more easily test further components, a configurable program was created to generate and train compatible artificial neural networks, using only the ReLU function and dense layers. The implementation was made using the Python 3 programming language, the Tensorflow framework, employing the neural network library Keras. The code implements a simple sequential, feed-forward neural network with a customizable number of dense layers, each with a customizable number of neurons. Then, the program imports and reads two datasets in the CSV (Comma-Separated Values) format, corresponding to the training and test sets. The model is trained through the Keras lirary's Adam optimization algorithm. With this implementation, the dataset can be changed to test cases with different complexities.

To use as an input for the prototype's next stages, a neural network was designed to implement a classifier for the Iris dataset (DUA; GRAFF, 2017). This dataset is relatively simple, suitable for testing proof of concept systems, with only four numerical variables, but it is well recognized in the pattern recognition scientific community and literature, cited by several articles, with a shortened list available in the dataset's website (DUA; GRAFF, 2017), and represents a more realistic and practical dataset. This database has 150 samples, each with 4 input variables, which are real numbers, and an output, which is a classification in one of 3 different classes.

Then, the dataset output variable was converted into the one-hot encoding representation. The resulting dataset has 3 binary outputs, one for each possible classification result. This method increases classification precision and distinction between the different results, and is compatible with the low number of possible output values. The database then had the order of its samples randomized and divided in two groups, with 50 and 100 items, corresponding to the test and training sets. The sample order is randomized in an attempt to homogenize the two data subsets in order to preserve the original dataset proportions.

Finally, the neural network has an input layer with 4 inputs, corresponding to the 4

dataset inputs, a hidden layer with 3 neurons, followed by 3 hidden layers with 5 neurons, and an output layer with 3 neurons, corresponding to the one-hot outputs. Every neuron implements the ReLU activation function, and every layer is fully connected. The training was executed until a 90% inference accuracy in the test dataset was obtained (5 classification errors in 50 test samples). This accuracy was deemed reasonable, since it demonstrates basic network functionality. While more optimizations are possible to further increase precision, these are not necessary to demonstrate the execution of the prototype's following stages. Additionally, these optimizations can increase the complexity of the neural network model, or include different technologies, which is against the rule to use only the ReLU function and dense layers.

4.2 Neural network data extraction

It is necessary to extract several neural network parameters to implement its inference in a customized hardware accelerator. As such, part of the prototype is the extraction of this data.

Since the artificial neural network used as input for the prototype was impmlemented using the Tensorflow framework and the Keras artificial intelligence library, the most standard and direct solution is to use the libraries' own export functions. Consequently, the model data was exported and saved via the SavedModel format, using the Keras' methods Model.save and models.load_model.

To export the model, the program that generates and trains the neural network uses the method Model.save to export the network's data. Note that this simulates the behavior of the accelerator generator's final user. Then, the generator system uses the Models.load_model method to import the necessary data from the user's network.

As such, the network's data corresponding to the topology, inter-neuron connections, number of layers and neurons in each layer, weights, biases and activation functions are available to the system. This information is sufficient to mirror the neural network's behavior and implementation, preserving the same results.

This step was executed and tested through the Microsoft's integrated development environment Visual Studio 2019 Community Edition's debug tool. During testing it was

possible to inspect the imported Model objects and ascertain that all of the necessary data was present, correct and accessible.

4.3 Activation function programming in VHDL

A crucial part of the final system is its function library, which consists in functional blocks of HDL code corresponding to different activation functions and neural network topologies. This library is accessed by the interpreter, which, after importing and reading the model data, samples and modifies the blocks in order to implement the necessary behavior to mirror the original neural network. An ideal library should contain as many different functions and topologies as possible, to be compatible with a wider array of models. However, as a prototype to examine the workflow to create the system and to diagnose possible problems with the current approach and demonstrate its viability, a proof of concept consisting in a single activation function and one network topology was considered sufficient, as long as the methodology supports the implementation of more functions in the future. The activation function chosen to be implemented is the Rectified Linear Unit (ReLU), which matches the function used by the neural network generated and trained previously. Another consideration is that this implementation targets each neuron individually, and in a higher hierarchy block, a separate component corresponding to every neuron in the network must be instantiated.

The HDL language chosen was VHDL. The ReLU function was implemented using the 32 bits floating point arithmetic functions and representation formats from the IEEE.float_pkg package, included in the VHDL-2008 version. This choice was made in order to completely preserve the original neural network behavior, as in software the bit-depth data representation format used is also 32 bit floating point numbers. Firstly, the neuron behavior is implemented through the multiplication of each of its inputs by its respective weights. Then, the results are summed and the layer's bias is added. Lastly, the resulting value is rectified, that is, if it is less than zero, the output becomes 0. Otherwise, the output is the addition's result. Beyond the ReLU function's implementation, the block also contains descriptions of input and output control signals. The first is the binary input "reset" signal, which resets the block if its value is one, and does nothing otherwise. The other control input is the "enable" signal, which is an 8 bit vector

that corresponds to an address respective to the system's data inputs. This signal is propagated through the system's pipeline to eventually match the resulting output to its originating inputs. Furthermore, as output control, there is the "done" signal, which corresponds to the "enable" value respective to the data output, that, when a non-zero value is outputted, indicates that the result is ready and which address it corresponds to.

The ReLU functional block was tested via its isolated simulation in the Intel's ModelSim software, using a range of possible input values and testing many different expected behaviors of the ReLU function. Examples are its overflow and underflow treatment, the rectification of negative outputs and the correctness of positive results, as well as testing the propagation of the "enable" signal through the "done" signal, and the signal timings. The test was considered successful, since the control signals were functioning correctly, propagating the input address and were correctly timed. Furthermore, the data resulting from the functional block was numerically correct.

4.4 Neural network topology programming in VHDL

The following step of the prototype is the functional block corresponding to the neural network topology, also belonging to the system's library. This block will be responsible for connecting the different neurons according to rules determined by the trained input model. Furthermore, the code must be written in such a way as to enable its posterior modification to include different neuron connection rules, and to be conveniently edited by the generator system to implement different neural networks. For this prototype, a single network topology was implemented: the fully-connected or dense layer. This corresponds to every neuron's output from the previous layer being connected to every neuron in the current layer. This choice also matches the topology chosen by the neural network generated and trained in the previous steps.

This block was also programmed in VHDL. The topology was implemented importing a generic ReLU neuron, which has multiple generic parameters that can be edited in their instantiation. However, in this prototype, since each functional block corresponds to a single neuron inside each layer, some parameters such as number of inputs are hardwired and cannot be edited during the component instantiation.

As such, for each different layer, a modifiable ReLU neuron is implemented. While it uses generic mapping functions to write necessary data, such as values corresponding to the weights used, the number of inputs and outputs remains fixed for each layer. Then, the neuron components corresponding to the neural network neurons are generated. Finally, using port mapping functions, the inputs and outputs of the neurons are connected following the topology rules, the input layer's inputs are connected to the network's inputs, and the output layer's outputs, to the network's outputs.

This functional block was tested using the previously programmed ReLU neuron functional block as component, and using a testbench which provided sample inputs as stimuli. Via simulation in the ModelSim software, it was possible to verify the correctness of the neural network behavior and that the internal signals were being correctly mapped.

4.5 Neural network accelerator generator programming in Python

The prototype's final step is programming a software which generates the VHDL code corresponding to the hardware accelerator which implements the artificial neural network.

This step uses the neural network generated and trained in the first step; extracts the relevant data from the model using the method described in the second step; and finally, writes VHDL code using the functional block library that includes the blocks programmed in the third and fourth steps.

The resulting prototype system, corresponding results and hardware accelerator performance was analysed to provide insights of further development and diagnose potential problems with the approach employed.

Many different metrics can be used to analyse the results, and these must be chosen and evaluated according to the context in which the system resides: low-power hardware accelerators for edge computing for Internet of Things (IoT) devices. As a consequence, a crucial metric to be analysed is the energy expenditure of the resulting hardware accelerator, as IoT devices usually have restricted power availability. Furthermore, according to the chosen implementation platform, synthesis on an FPGA,

the hardware must fit in a device compatible with the IoT environment. That is, the amount of integrated circuit components required to synthesize the design must be compatible with a small, low-power FPGA designed for this type of application. Then, the hardware must be evaluated in regards to its execution performance, that in this case can be assessed by measuring its inference result throughput. Finally, depending on the technologies used, there may be model accuracy loss with its implementation in hardware, and this loss should be measured and taken in consideration.

All of these metrics were considered to diagnose problems with the implementation and approach used by the prototype, and the issues recognized were solved to develop the functional proof-of-concept.

The VHDL hardware accelerator generator was implemented in the Python 3 programming language. The generator begins by importing the neural network exported model. Since the prototype's neural network was developed using the Keras library based on the Tensorflow framework, their import methods were used, generating a Tensorflow model file. In this prototype, the file path must be directly referenced by editing the program's code, and while it imports the model successfully, it must be addressed to increase ease of use by a final user.

Once the model is imported, it is read iteratively along the neural network layers, and for each layer a function is invoked to write a VHDL file corresponding to the kind of neuron used. In the network used by the prototype, this stands for three different neuron VHDL files, with 4, 3 and 5 inputs, and all of them implementing the ReLU activation function.

While the model is read to extract the neuron data, network parameters are also stored, which are number of layers, neurons per layer, interconnectivity rules and number of network inputs and outputs. When the program finishes reading the model, another function is invoked to write the VHDL file corresponding to the network's topology following the parameters extracted. The topology file imports the neuron's previously generated corresponding component files, generates the correct amount of each neuron, and connects them following the specification.

Since every layer in this example is dense, the outputs from all neurons from a layer are connected to the inputs of every neuron of the following layer. Finally, the network's

inputs are connected to the first layer, and the final layer's outputs are connected to the network's outputs.

In the context of compiling the VHDL code, the topology file is hierarchically the the top entity, representing the entire neural network.

The generator system's operation was tested through the generation of the VHDL code corresponding to the previously mentioned trained Iris dataset classifier neural network. VHDL files for each neuron were properly generated, as well as a topology file. A logic simulation test was executed using the ModelSim software, using a test-bench file which provided the 50 samples from the dataset test input data as the 32 bit floating point binary vectors used by network as stimuli. Furthermore, the stimuli file included "enable" signals along each input data to label them, as well as "reset" and "clock" signals for the device's operation.

The system was then successfully simulated, and its behavior was as expected. Since there is a single hardware pipeline in the accelerator, with each stage corresponding to each network layer, each sample's inference result is ready after 5 clock cycles from its input, since there are 5 layers in this network. The results are correctly labeled with the representing "enable" signal which accompanied its input being outputted in the "done" signal. The output data results were exactly the same as the results obtained during the software tests, which means that the hardware implementation incurs no loss in accuracy when compared to the original network. This zero accuracy loss was expected, since the floating point arithmetic used in software is mirrored in the VHDL files, as well as using the same precision in the data representation.

Finally, the resulting VHDL files were tested in a synthesis and mapping context for the implementation of the accelerator in an FPGA. The FPGA choice must be made considering the projects requirements concerning the use of the hardware accelerator in an edge computing for an Internet of Things device. As such, the chosen FPGA is an ultra-low-power, with small circuit area and especially low number of internal components, the Lattice Semiconductor's iCE40UP5K-UWG30. To synthesize the hardware in this FPGA, the Lattice Semiconductor's iCEcube2 software was used, configured to target the specified FPGA, with no other design and compilation restrictions.

While the compilation and synthesis process was concluded with no errors, the im-

plementation and mapping was unsuccessful. The resulting reports from the software pointed to two fundamental problems. The first of which was that the FPGA critical path analysis diagnosed that the clock slack obtained was -69,212 ns when attempting to use a 1MHz clock. This means that in the worst circuit pathway in the resulting hardware, the system could not deliver its results to the next step in the time given by the 1MHz clock, being late by -69,212 ns and the hardware would not function as intended. The resulting hardware implementation would only reach a 0.9MHz performance. The second issue is that the use of internal components greatly exceeds the available number of components available in the chosen FPGA, using more Lookup Tables and adders as it would be possible.

4.6 Prototype discussion and conclusions

The artificial neural network used as sample input for the generator system was considered to be successfully designed, generated and trained. The dataset used has relevance and recognition in the pattern recognition and artificial intelligence fields of research, had its classifier implemented adequately, with acceptable topology and accuracy performance. Furthermore, the artificial neural network was generated according to the prototype's restrictions of topology and activation function choices. Note that as the system's libraries are expanded to include more functions, more complex networks will also be necessary to test them.

The parameter extraction was successful, as the neural network model was exported and imported into the generator system with all of its necessary data. However, the prototype is only compatible with the Tensorflow framework, using the Keras library. Since increasing framework and library compatibility is a main concern of this system, it would be important to the generator's success to include more readable formats. An efficient and effective way to include multiple different formats with minimal work would be to program compatibility with the ONNX (Open Neural Network Exchange) format (BAI et al., 2019). The ONNX project is an initiative whose objective is the common representation of many different machine learning models among a variety of frameworks, toolsets, libraries and compilers. It is also open source, and already boasts several open converters from different, relevant formats into the ONNX exchange for-

mat, such as Tensorflow, Caffe, scikit-learn and many others. As such, if the hardware accelerator generator system is programmed to be compatible with ONNX, it will also be indirectly compatible with every format that can be converted into ONNX using an open converter.

The ReLU activation function was correctly implemented in VHDL considering its logical behavior, as the results outputted when the accelerator is tested in simulations are the same as when the network is tested in software. However, this initial implementation has its issues, namely in the complexity of the resulting circuit that is necessary to synthesize this behavior in hardware. The consequence of this complexity is that the final hardware requires a high number of integrated circuit components for its final synthesis, and the resulting clock frequency is low.

Many characteristics of this implementation contribute to this. One of which is that the floating point addition and multiplication functions used are implemented through a VHDL library that synthesizes the arithmetic operations without considering pipelining techniques. This results in a final circuit that executes the entire operation in a single clock cycle, serializing the internal circuit components. This serialization reduces the clock frequency, as the clock is required to wait for the entire operation to complete. Furthermore, the internal circuits are poorly utilized, as each serial step is idle, waiting for the previous component to complete its operation to begin computing. With the resulting long serial line of components, there is low efficiency as many steps are left waiting. Another issue with the prototype's implementation is that all the arithmetic operations in each neuron are sequential as well, also incurring in the aforementioned problems.

To solve these issues, the final accelerator architecture include several pipelining steps. The critical paths and complex operations were diagnosed, and separated into stages, greatly speeding up clock frequency. In addition, the different stages can work simultaneously, with results from different inputs being computed in parallel, increasing the circuits efficiency, reducing idle times and speeding up inference results.

Furthermore, while we observe that, since the entire network's topology and arithmetic operations were mirrored in hardware using the same value representation precision, which result in zero inference accuracy loss, this also results in highly complex

synthesized hardware. Working using 32 bit floating point accuracy greatly increases FPGA resource use, and even while testing a simple network, this makes synthesis impossible. As such, reducing value representation precision was chosen as a strategy to reduce circuit complexity, and following Tensorflow Lite's quantization algorithm for integer-arithmetic-only inference (JACOB et al., 2017), the incurred precision loss is minimal. As a result, the final accelerator generator uses quantized neural networks to an 8 bit precision, which are trained taking the precision loss into account, with greatly simplified and reduced resulting hardware and minimal accuracy loss, which enabled hardware synthesis in devices compatible with the Internet of Things context.

5 ACCELERATOR ALGORITHMS

The final hardware accelerator is composed of several algorithm implementations in hardware description language code that are stored in the generator library. For a proof-of-concept implementation that would properly demonstrate the system's functionality and viability, a minimum viable set of algorithms was chosen to be implemented.

The algorithm set to be implemented must include an activation function to be executed by the neurons and an algorithm to implement a neural network topology to execute the basic functionality of a neural network.

Additionally, as diagnosed in the prototype's intermediary results, considerations must be made to solve the issue of high computational resource demand, and this was tackled with the inclusion of variable quantization algorithms.

5.1 Variable quantization

As diagnosed in the prototype's results and discussion, the 32 bit floating point variable representation is significantly cumbersome and demanding of computational resources, both in integrated circuit demand to implement the operations and time to execute them. As such, we chose to implement a variable quantization scheme to reduce variable representation bit depth to simplify the resulting hardware and inference operations at the cost of result accuracy. The quantization methodology chosen is the Tensorflow Lite's full integer quantization for integer-arithmetic-only inference (JACOB et al., 2017).

As demonstrated in Jacob et al. (2017), when paired with quantization aware neural network training, the inference accuracy loss is minimal. This methodology was chosen as the prototype concluded that full 32 bit precision was prohibitive to the accelerator's synthesis and implementation in small FPGAs compatible with Internet of Things devices, and the accuracy loss is acceptable.

The 8 bit integer variable quantization uses the following formula (JACOB et al.,

2017):

$$real\_value_{\alpha} = (quantized\_value_{\alpha} - zero\_point_{\alpha}) \times scale_{\alpha} \tag{5.1}$$

which can be more conveniently rewritten to match the following matrix notations as:

$$r_{\alpha}^{(i,j)} = S_{\alpha} \times (q_{\alpha}^{(i,j)} - Z_{\alpha}) \tag{5.2}$$

Note that in 5.1, 5.2 each variable α is shifted by a zero point and scaled by a scale, and these parameters are the same for all the elements in the same array. While the zero point is an integer, corresponding to a shift in the new 8 bit representation, the scale is a floating point number, as it converts the variable into a rounded integer via a multiplication.

These parameters are obtained during the neural network quantization-aware training, during which they are trained along the other neural network weights to minimize inference error. Once trained, the quantization parameters are fixed and are not modifiable during neural network execution.

Each of the quantized weight, bias, input and output arrays of each of the layers are quantized and have their own parameters.

Note that the result of the multiplication between the 8 bit weight and layer input arrays result in a 32 bit integer array. Since biases are added after this multiplication, they can be represented as 32 bit integers to benefit from the increased precision before their result is cast down to 8 bit integers for the next layer.

5.2 Layer topology: Matrix Multiplication

The first of which consists in a matrix multiplication functional block. This functional block is necessary as it represents the fundamental operation executed to perform the fully-connected layer neural network topology, which was the layer topology chosen to demonstrate the system. The fully-connected layer behavior consists in each of the neuron's inputs being multiplied by their respective weights stored in the layer and these results are added together to form the layer's activation function's input. Each neuron has its own individual weights for each of the inputs. Furthermore, the layer can optionally include a bias value, which is also added to the input-weight multiplication

result.

This behavior is implemented via a matrix multiplication hardware functional block. In the functional block, each of the layer's inputs are stored in an one-dimensional input array. The layer's weights are represented by a matrix, implemented as a two-dimensional array. The entire layer's weights are grouped together in this array, with each neuron's weights being represented as a column in the matrix, while the rows represent the weights corresponding to each specific input.

The multiplication of the neural network's inputs by each of the neuron's weights is then executed by the matrix multiplication of the input array by the weights matrix. The result is a one-dimensional array, with each item corresponding to the activation function input for each of the neurons in the neural network.

Note that the entire layer's operation is grouped together in this multiplication, with every neuron's execution being represented by one of the dimensions in the weight and output matrices. Another consideration to be made is that the inputs, weights and results in this stage are quantized and must be corrected before being used, whether in calculations in further layers or as neural network outputs.

The matrix multiplication is then implemented by first multiplying each weight matrix row by the input array and summing the results, and iterating this operation over the remaining weight matrix rows. Once the addition for each row is complete, if the layer includes the optional bias, it is then added to the result and outputted.

5.3 Activation function: Rectified Linear Unit

To demonstrate a successful neural network implementation, an activation function to be executed by the neurons must be implemented. As discussed previously, the Rectified Linear Unit (ReLU) function was chosen to be implemented.

The ReLU function behaves as follows:

$$ReLU(x) = \begin{cases} x & \text{if } x \geq 0, \\ 0 & \text{otherwise.} \end{cases} \tag{5.3}$$

As such, the activation function implementation receives an one-dimensional array corresponding to the layer topology's results. Then, a loop checking whether or not the values in the array are positive, simply outputting the value if it is, and outputting zero otherwise.

However, since the neurons are using the quantized arithmetic representation, a zero-point correction of a variable may be in use. This means that zero may be represented as a different number depending on the quantization parameters being used by each variable. As a consequence, the ReLU function's threshold must be configurable to each variable's quantization parameters. Since the quantization parameters are fixed once the network is trained and deployed, and don't change during inference with different inputs, the ReLU threshold can be configured and hardwired at synthesis. As such, the final ReLU implementation behaves as follows:

$$ThresholdedReLU(x) = \begin{cases} x & \text{if } x \geq zeropoint, \\ 0 & \text{otherwise.} \end{cases} \tag{5.4}$$

Note that in 5.4, $zeropoint$ corresponds to the value that represents zero in the activation function's input variable.

5.4 Layer quantization corrections

Each layer in the neural network operates under its own variable quantization parameter criterion, with its inputs, outputs and intermediary variables having their own quantization parameters. To properly implement the layer's behavior in hardware, the results of each layer must conform with the quantization rules of its output. As such, once the quantized arithmetic operations are completed, quantization corrections must be applied to the numerical results obtained to conform with the output parameters. This guarantees that the values are readable by further layers and behave as expected.

As a consequence, we must determine the corrections necessary to maintain arithmetic result correctness by following the operations executed with the quantized variables.

5.4.1 Quantized matrix multiplication

Because of the way quantized values are represented in Equation (5.2), arithmetic involving these values must be adapted to produce the expected results. From the definition of matrix multiplication and the quantization representation specification (5.2), we have:

$$S_3(q_3^{(i,k)} - Z_3) = \sum_{j=1}^{N} S_1(q_1^{(i,j)} - Z_1) \times S_2(q_2^{(j,k)} - Z_2) \tag{5.5}$$

which can be rewritten as

$$q_3^{(i,k)} = Z_3 + M \sum_{j=1}^{N} (q_1^{(i,j)} - Z_1) \times (q_2^{(j,k)} - Z_2) \tag{5.6}$$

$$M := \frac{S_1 S_2}{S_3} \tag{5.7}$$

In the neural network context, q_3 corresponds to the quantized layer's outputs; q_1 are the quantized layer's inputs; and q_2 are the quantized layer's weights.

Furthermore, as the input array only has a single dimension, $i = 1$ and j is in the interval $[1, inputs]$, with $inputs$ being the number of inputs in the neural network layer. Similarly, k is in the interval $[1, neurons]$, with $neurons$ being the number of neurons in the layer

In addition to the matrix multiplication between layer inputs and weights, the optional bias is then added to the result:

$$q_3^{(i,k)} = Z_3 + S_{bias}(q_{bias} - Z_{bias}) + M \sum_{j=1}^{N} (q_1^{(i,j)} - Z_1)(q_2^{(j,k)} - Z_2) \tag{5.8}$$

However, the bias' quantization parameters in the Tensorflow Lite's 8 bit quantization are conveniently defined as:

$$S_{bias} := M \tag{5.9}$$

$$Z_{bias} := 0 \tag{5.10}$$

and as such, Equation 5.8 can be simplified to:

$$q_3^{(i,k)} = Z_3 + M(q_{bias} + \sum_{j=1}^{N}(q_1^{(i,j)} - Z_1)(q_2^{(j,k)} - Z_2)) \tag{5.11}$$

5.4.2 Efficient handling of zero-points

To efficiently implement the zero-point sums and subtractions(JACOB et al., 2017), Equation (5.11) is rewritten as

$$q_3^{(i,k)} = Z_3 + M\Big(q_{bias} + NZ_1Z_2 - Z_1a_2^{(k)} - Z_2a_1^{(i)} + \sum_{j=1}^{N} q_1^{(i,j)}q_2^{(j,k)}\Big) \tag{5.12}$$

where

$$a_2^{(k)} := \sum_{j=1}^{N} q_2^{(j,k)}, \qquad a_1^{(i)} := \sum_{j=1}^{N} q_1^{(i,j)} \tag{5.13}$$

Next, following TensorFlow Lite's specification (JACOB et al., 2017), the zero-point for the weights matrix on fully-connected layers, Z_2, is always 0. As such, the terms NZ_1Z_2 and $-Z_2a_1^{(i)}$ are also always equal to 0. Additionally, $a_1^{(i)}$ doesn't need to be computed, and $a_2^{(k)}$ can be computed offline during compilation, saving time and resources during inference.

The final matrix multiplication equation is then:

$$q_3^{(i,k)} = Z_3 + M\Big(q_{bias} - Z_1a_2^{(k)} + \sum_{j=1}^{N} q_1^{(i,j)}q_2^{(j,k)}\Big) \tag{5.14}$$

This final equation means that once the matrix multiplication and addition for each of the output's elements is complete and the quantized bias is added to the result, the quantization corrections necessary are:

1. Add $-Z_1a_2^{(k)}$, which is calculated offline during compilation;
2. Multiply by M;

3. Add Z_3.

5.4.3 Efficient handling of multiplication by M

As described in 5.1, the scale quantization parameters are floating point numbers. As a consequence, the variable M, defined by 5.7, is also a floating point number. While its computation can be done offline since the scale values do not change during inference, floating point multiplications are significantly resource intensive.

However, we can rewrite M as follows:

$$M = 2^{-n} M_0 \tag{5.15}$$

where instead of multiplying by a floating-point number, we multiply first by the fixed-point normalized parameter M_0, which is in the interval $[0.5, 1)$, and by then multiplying by 2^{-n}. n is defined as a positive integer.

The multiplication by a fixed-point parameter is significantly faster and less computational resource intensive than floating-point arithmetic. Furthermore, multiplication by a power of 2 can be efficiently implemented in hardware by a bit shift with round-to-nearest behavior.

Depending on the exact instruction or implementation used for bit shifts, the resulting algorithm can include an asymmetric bias in relation to zero, which has been shown (JACOB et al., 2017) to cause significant loss of accuracy in neural network inference. An example of bias would be with instructions that always round numbers downwards, towards negative infinity. This incurs a downward bias that skews neural network training and inference. This loss is avoided when using a bit shift with a symmetric round-to-nearest behavior. To implement this rounding rule, the following correction to shift a signed value by n bits is used:

1. Add 2^{n-1} to the value being shifted;

2. Invoke the bit shift by n

To calculate M_0 and n, we multiply M by 2^n, starting from $n = 0$ and iteratively

increase n by 1 until the result, M_0, is in the interval $[0.5, 1)$. This is done by the accelerator generator program and the result is stored in a hardware description language data package. This calculation doesn't impact inference performance as it is done during the hardware generation.

6 GENERATOR SYSTEM

Following the conclusions drawn from the prototype and its intermediary results, the final system design was made in order to solve issues detected and preserve its strengths.

The final system still follows the same general structure as the prototype: it is divided between the generator system and the generated accelerator. The generator consists in a software program which is used by the final user to read and interpret an artificial neural network and automatically generate code that can be synthesized to implement a hardware accelerator that executes said network. The generated accelerator, on the other hand, is the hardware description written by the previous system.

In this chapter, we discuss the generator system's structure and relevant design choices made.

The main objectives of the generator system's design choices are maintained from those of the prototype.

- The resulting software must be usable by end users that do not have specialized hardware programming knowledge;
- The program must be modular and expansible to facilitate its longevity and continuity;

To accomplish the first design objective, the generator program does not need any input from the user other than the trained neural network model, isolating specialized knowledge. As for the second objective, the hardware code generation is based on consulting and modifying a modular hardware functional block library. This library can be expanded and is divided between activation functions and network topology, each with a set of design rules to ensure compatibility.

The generator system consists in a Python application, developed and tested using Microsoft's Visual Studio integrated development environment.

As shown in figure 1, the generator can be divided in 4 main sections:

1. Neural network data import;
2. Neural network data extraction and interpretation;
3. Neural network hardware module library;
4. Hardware description language writer.

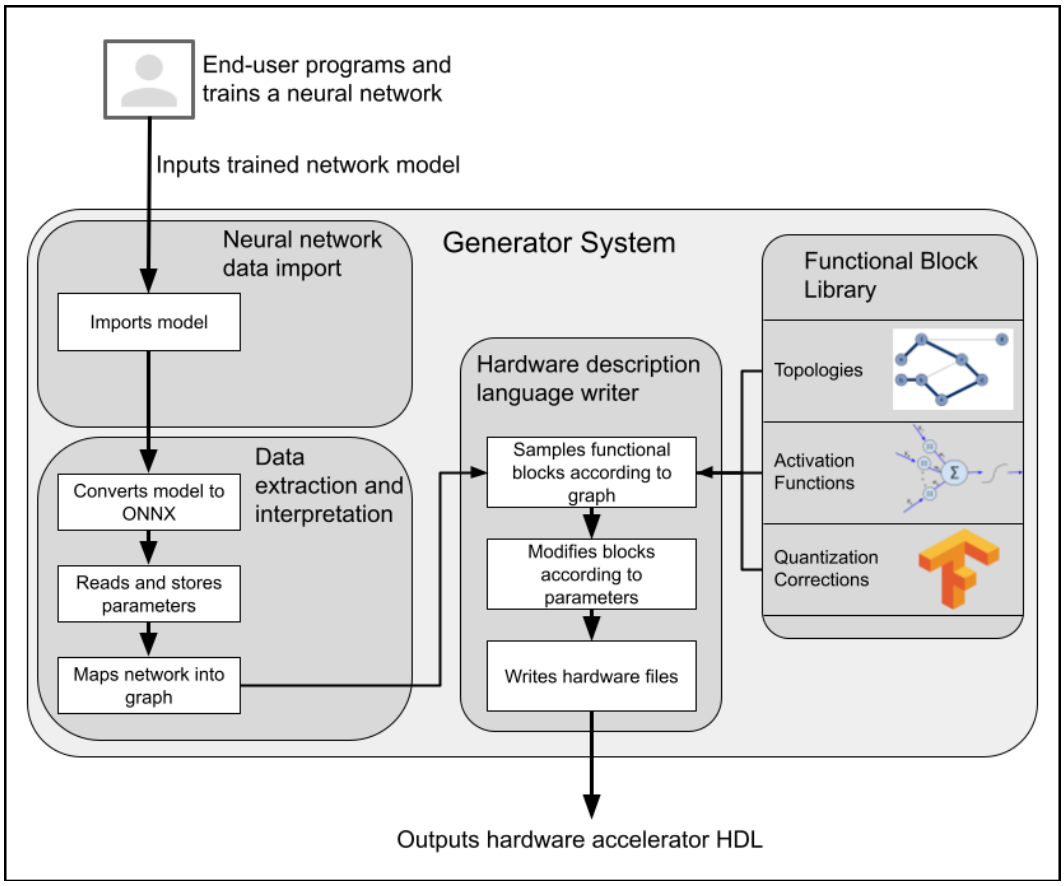


Figure 1 - Generator system structural diagram

6.1 Neural network data import

The neural network data import section of the generator software consists in the part that receives and reads the neural network model files. At the moment, the system file import is only compatible with models developed and exported by Tensorflow Lite's framework for neural network development. Note however, that while the code to import the files is simple and framework dependant, framework compatibility issues are tackled by the next step in the generator system, which uses conversion into the ONNX format.

This step consists in a simple graphics user interface, made with Python's native GUI library, that asks for a compatible neural network file and loads it into the program.

6.2 Neural network data extraction and interpretation

The second step in the generator system is the neural network data extraction and interpretation.

To maximize machine learning framework compatibility, before interpreting models, we first convert the neural network into the Open Neural Network Exchange (ONNX) format (BAI et al., 2019). ONNX is a project by Facebook, Microsoft and the Linux Foundation to create an open-source artificial intelligence format to establish common standards and communication between different machine learning frameworks and tools. The format resulting from this project is a readable graph that accurately describes the topologies, functions and other training data from the neural network, and can be used by the generator system to interpret the models received.

The main advantage in using the ONNX format is its vast library of open-source converters available to use. Each of these converters can receive neural network models from a development framework and convert it into the ONNX format. As such, as our generator system is compatible with ONNX, the system is indirectly compatible with every other framework that can be converted into ONNX by one of the open-source converters with minimal changes to the data extraction and interpretation software. This greatly increases system compatibility with artificial intelligence frameworks and technologies.

Once the model is converted, a graph is created in Python to store the neural network elements, described by nodes in the ONNX format. Additionally, dictionaries are created to store data pertaining to the operations executed by each node and to what nodes the inputs and outputs of each node go to. The graph and dictionaries are populated by iterating along the nodes in the ONNX graph.

Once this is completed, lists of every activation function and network topology used are composed in order to be used by the next step in generation, to inform which modules to sample from the module library. This list matches the operation names that

follow ONNX's naming conventions to the specific hardware models available in the function library.

6.3 Neural network hardware module library

The hardware module library consists in a collection of .py package files that are imported by the main Python program. These files are created by first programming the function to be executed in VHDL, with every internal changeable variable being editable and stored in VHDL generics. Generics are a form of local constant that can be assigned values during the component instantiation. This means that another VHDL configuration file higher in hierarchy can call a component file and configure it with relevant data matching a neural network. For example, when synthesizing a densely connected neural network layer, the layer parameters can be programmed as generics, and as such, are easily editable by the generator program.

As a result, the hardware accelerator is modular and the generator system can edit the component files with the data read from the imported model with little difficulty. However, the VHDL components must be programmed with this design methodology to ensure compatibility and facilitate the automatic generation.

The generator system uses the lists organized by the previous step and calls for print functions imported from the python packages corresponding to the functions and topologies used. Note that due to the design using generics, the files are printed as-is, with its variables being edited during compilation using a configuration file. This facilitates readability, as the Python program prints the files directly. This choice also facilitates module testing, as sample values are configured into the generics, and while not used during the actual hardware synthesis, they can be used to test the individual components. Furthermore, this standardizes a pattern to be followed by contributors, helping the system's longevity as an open-source project and ensuring internal compatibility.

6.4 Hardware description language writer

The writer section of the generator system invokes the print functions from the required packages, imported following the lists composed by the model interpretation step. Additionally, print functions related to core VHDL files are also called, such as a clock frequency divider module, necessary to speed up clock frequency, and the top hierarchical datapath file that organizes the entire accelerator.

Other than the files corresponding to the different topology and activation function modules, there is also a network_data_package.py file corresponding to the network data package file composition. This is the only file edited during the program's runtime, and it includes every neural network internal variable specific to the imported model. This includes number of layers, neurons per layer, interconnectivity rules, variable bit depth representation, quantization parameters, weights and biases. When called by the writer section of the generator, this function also calculates parameters respective to the quantization variables that are computed offline to save inference time. Namely, the quantization parameters M_0, n and $a_2^{(k)}$ from the quantization correction modules of each quantized layer are calculated in this step.

Furthermore, the network data package function also translates the variables read in the ONNX format into signed binary arrays readable by the rest of the hardware. Note that to maximize compatibility and due to VHDL's incompatibility with jagged arrays and difficulties with modifying port sizes during synthesis, weights and biases are concatenated and vectorized, requiring special treatment to be accessed by the hardware components. For example, an array of 5 8-bit values is represented by a single, 40-bit vector with the values concatenated. Furthermore, unused parts due to the resulting vectorized array being jagged are filled with zeros, and should be automatically optimized by the VHDL compiler during synthesis. Note that the arrays are jagged as a single array stores information for multiple layers, and the number and size of the values changes according to parameters such as amount of neurons in the layer.

Once the modifications are made, the .vhd files are written and outputted, ready to be simulated to check for correctness, or synthesized to be implemented.

6.5 Expanding the generator system

One of the project's main objectives is that the resulting tool can be expandable. This expansibility can take different forms: new neural network algorithms can be included, such as new activation functions or topologies, or new machine learning frameworks can be included in the system's compatibility.

The first consideration in order to ensure expansibility is that the software is published online under an appropriate open source license. This has been done by publishing the code and commentaries using GitHub's internet hosting system for software development, which allows for hosting and sharing code repositories for free, under a chosen license (MUYAL, 2022). The system is briefly described in the GitHub repository, with instructions for its use and the inclusion of new functions and topologies.

In turn, the open source license chosen is the GNU General Public License v3.0. This license allows for third parties to do almost anything with the software, including distributing, modifying, and using for private and commercial use. The conditions are that when the software is distributed, the resulting source code must also be disclosed under the same license, documenting the changes made. This ensures the open source project's continuity, forbidding closed source versions and encouraging contributions. Furthermore, the license provides limitations of liability and warranty, with the software being provided "as is". As such, the license chosen is highly permissive, with limitations that protect the developers and forbids distribution of closed source versions to encourage the system's continuity as an open source project.

6.5.1 Including new activation functions

Conforming to the pattern used by the Rectified Linear Unit activation function implemented, the following is necessary:

- A .py file that prints a VHDL file that implements the activation function;
- Modify the qNPUInterpreter.py file to include the new operation in the "synthesizable_operations" list;

- Modify the qNPUInterpreter.py to include the new print file, and to call the new print file if the new operation is found in the interpreted model.

Note that the activation functions are used as components, being called by a VHDL entity higher in hierarchy. Additionally, relevant internal variables, such as the rectifying threshold in the Rectified Linear unit, are to be coded as generics, so they are easily modifiable by the top-level entity, and configurable by the network data package file.

6.5.2 Including new topologies

The implemented neural network topology, the fully-connected or dense layer, is calculated using the matrix multiplication to provide the weighted inputs to the layer's neuron's activation functions. In order to include different topologies, such as convolutional layers, different neural connectivity rules must be implemented instead of the full matrix multiplication. For example, in the case of the convolutional layer, a shifting window over the previous layer's outputs is to be considered as inputs for each of the layer's neurons.

As such, to include new neural network topologies, the following is necessary:

- A .py file that prints a new .vhd file that corresponds to the new connectivity rules;
- The printDatapathv2.py file must be modified to include the new topology file;
- The qNPUInterpreter.py must be modified to detect the new topology and call printDatapathv2.py accordingly.

6.5.3 Including new frameworks

As cited previously in 6.2, to maximize compatibility with machine learning frameworks and technologies, the generator system converts the received neural network models into the ONNX format (BAI et al., 2019). As such, by being compatible with ONNX, the system is indirectly compatible with the frameworks that can be converted into ONNX using one of the many open source converters available.

Consequently, to include new frameworks, first check ONNX's website for the supported tools and available converters. If the framework to be included has a converter

available, follow the instructions and include the steps necessary in the qNPUInterpreter.py to import a model and convert it into ONNX. Once this is done, feed the ONNX model into the rest of the software and the interpretation should work with few minor changes.

If the framework is not supported by ONNX, a full converter must be implemented for compatibility with the interpreter, and is not recommended.

7 ACCELERATOR HARDWARE ARCHITECTURE

As cited in the literature review and used by the prototype, there are two main neural network accelerator design paradigms (PLAGWITZ et al., 2021): dedicated circuit design and overlay-based co-design. The prototype implemented an accelerator based on dedicated circuit design, that is, the full and direct implementation of the entire neural network structure in hardware, mirroring its original architecture. Since the Internet of Things edge computing environment is so restrictive, the best case scenario is fitting dedicated circuits into small FPGAs and exploiting their better performance and efficiency.

The prototype has a prohibitive overuse of available hardware resources, not allowing the hardware's synthesis in an adequately sized FPGA, and this conclusion could call for a paradigm change into overlay-based co-design, which uses FPGA resources more efficiently and predictably. However, the inclusion of the quantized arithmetic and other hardware design techniques to be described in this chapter were deemed sufficient and we chose to keep the better performance and efficiency provided by the dedicated circuit design choice.

As such, the resulting hardware accelerator architecture closely mirrors the topology of a neural network in software, as shown in figure 2.

Neural networks in software are divided by layers, connected sequentially, and the accelerator hardware follows this structure. Each layer is connected either to the previous layer's outputs or the network's inputs, and in turn sends its outputs to the next layer or the network's outputs.

The first optimization when compared to the hardware generated by the prototype is the inclusion of a register bank that stores the layer's inputs, as shown in figure 2.

This has many advantages, the first of which is to enable pipelining behavior in-between layers, as the previous layer does not need to hold the output value as it is stored near the hardware which requires this data. Without the register banks acting as memory, the values necessary for calculating the following steps would be lost with each clock cycle if they aren't specifically held by the previous step, which means that

each block that has an output would need to wait until their results are not needed anymore to continue execution. However, with the inclusion of memory, these values are stored locally where they are needed, freeing the hardware that generated the output. This means that, when processing multiple inferences for many sets of inputs, the previous layer does not need to wait for the other layers and can begin calculating the output for the next inference. As a consequence, the hardware spends less time idling, speeding up execution speed and efficiency. Furthermore, by relieving each layer of the responsibility of holding the output values, there is no longer the need to synchronize inputs and outputs among different layers, as they store the necessary data by themselves.

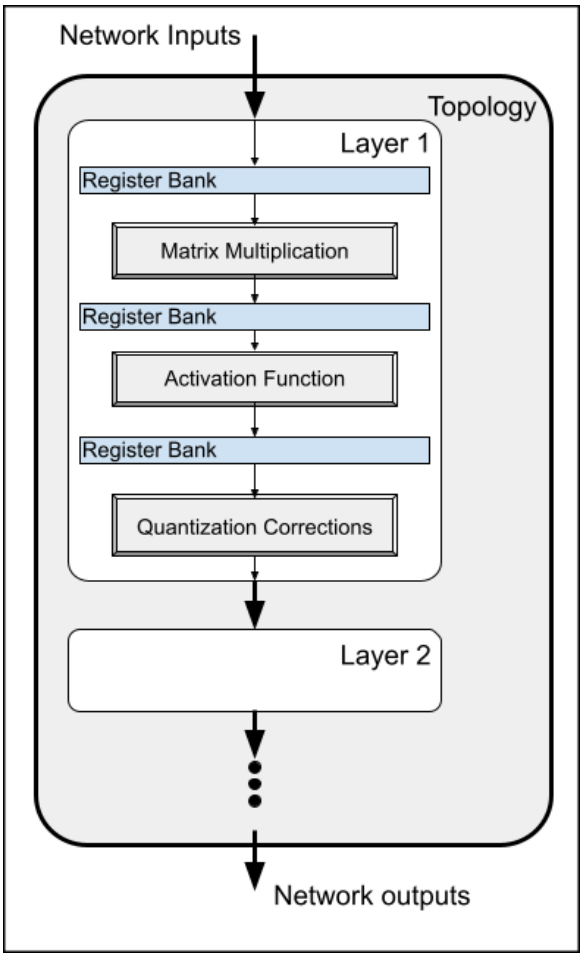


Figure 2 - Accelerator architecture diagram

Each neural network layer is composed by 3 different modules:

- A module corresponding to the neuronal interconnectivity rules;
- A module corresponding to the activation function to be executed by the layer's

neurons;

- In the case of layers that implement quantized arithmetic, a module that executes corrections related to the quantization.

Furthermore, there are register banks between each of these modules, further exploiting the advantages of pipelining and easier synchronization between modules.

The network's layers are organized by a top-level hierarchical entity represented by the network's topology. The first of the entity's responsibilities is to call for each of the necessary components to implement the neural network's layers. This includes each of the necessary models cited previously, the interconnectivity rules, activation functions and quantization corrections. In the case of the neural network implemented to exemplify the system's functioning, this corresponds to the matrix multiplication module for the fully-connected layer, the rectified linear unit as activation function, and the corrections pertaining to the full-integer-arithmetic quantized matrix multiplication. Note that both the matrix multiplication and rectified linear unit use quantized arithmetic, greatly increasing execution speed, and reducing operation complexity and hardware resource demand.

The second top level entity's task is to import the network data package. This is the file that is written by the generator system after the neural network's interpretation, and has all of the network's relevant variables. As shown in figure 3, the network data package includes structural data, such as the number of layers, the number of neurons of each layer and how many inputs each layer has, and the bit depth representation of the values of each layer. Furthermore, the data package includes the neural network trained values, the weights and biases used. In addition to the parameters shown by figure 3, the data package also has every quantization parameter necessary for the corrections and quantized arithmetic: the zero points for every array, the calculated M_0 and n parameters corresponding to the scales, and the precalculated $a_2^{(k)}$ arrays.

Lastly, the top level entity uses generation functions to instantiate the neural network components and interconnects the inputs and outputs of these components via port maps. Additionally, during the generation, the values stored in the network data package are used to configure the generics used in the modules, ensuring the trained parameters are used by the neural network accelerator.

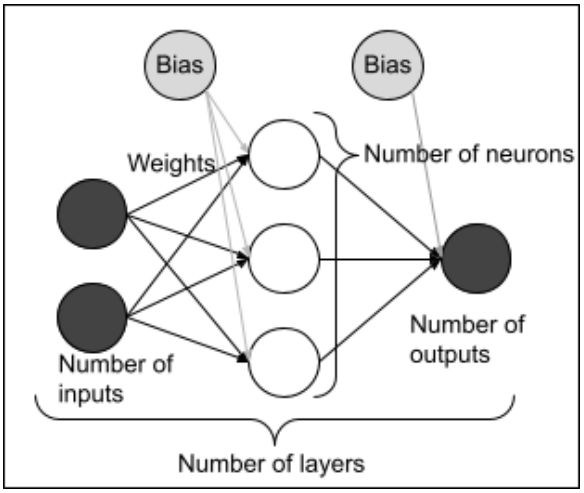


Figure 3 - Neural network parameters stored by the network data package

Another functional block implemented to maximize hardware compatibility and future-proof in relation to other different neural networks is the frequency divider used. This block consists in a simple circuit that divides the system clock's frequency depending on the value of a variable that can be edited by the generator system. As such, depending on the demands from different neural network designs and target devices for synthesis, this block can be edited to provide an internal clock frequency compatible with the operations necessary. For example, if the neural network to be implemented is highly complex, it is possible that the worst critical path is too long to be executed in a single clock cycle with the lowest frequency provided by the FPGA. In this case, the frequency divider can decrease the internal clock frequency for the worst critical path to be executed while relaxing demands to the device clock. As a consequence, this counteracts the problems that arise from the minimum possible device clock frequency and worst critical paths in different neural network circuits. Also note that the hardware implementation of a clock divider by a constant power of two can be simply implemented by the synthesizer using a single flip-flop and an inverter per power of two. This means that the implementation is simple and doesn't require many computational resources.

Another optimization included in the new version of the synthesized hardware accelerator is the use of loop unrolling techniques. When a loop is described in VHDL code using behavioral modelling, the compiler interprets the operations executed by the loop and decides the hardware necessary to implement the loop. However, when simply calling for a loop in the description, the required behavior from this description

is the execution of all of the loop's steps during a single clock cycle. This can cause a series of issues: the loop operations may be serialized, requiring a long critical path of hardware whose operations are to be executed in a single clock cycle. As a consequence, most of the hardware is idling, waiting for its turn in the sequence to execute its task, reducing the circuit's efficiency, and the system's clock frequency must be significantly decreased to allow for all of the operations to conclude.

This issue is avoided by unrolling the loops, that is, describing the hardware in a way that each loop iteration is executed in its own clock cycle. When combined with register banks and pipelining between important steps, loop unrolling greatly speeds up execution and clock frequency, as well as decreasing clock period overheads for the rest of the hardware that used to wait until the worst-case critical path concluded its operations to resume inference.

Furthermore, the quantization corrections of the matrix multiplication are optimized to include a pipeline with a finite state machine with multiple stages, each of which with its loops unrolled. Each of the necessary corrections has its own unrolled loop that takes its turn executing, and when done, progresses the finite state machine to the following state, corresponding to the execution of the next quantization correction. The stages correspond to the corrections detailed in the algorithms chapter, 5, with states pertaining to zero point additions, the efficient handling and multiplication by M_0 and round-to-nearest bit shift, and overflow corrections. Once the corrections are done, the data is ready to be outputted from the layer and either used by the following layers or outputted from the neural network itself.

7.1 Functional block hardware architecture

Each of the functional blocks, whether they correspond to the implementation of matrix multiplications for neural network topology, neuron activation functions or quantization corrections, follow a similar design pattern. This pattern aids the system's expansibility. As long as new functional blocks are implemented following the same pattern, they should be compatible amongst themselves. Namely, the compatibility issues that could arise and are solved by following the design pattern are synchronization issues between multiple blocks, and data formats expected as inputs and outputs.

7.1.1 Synchronization between functional blocks

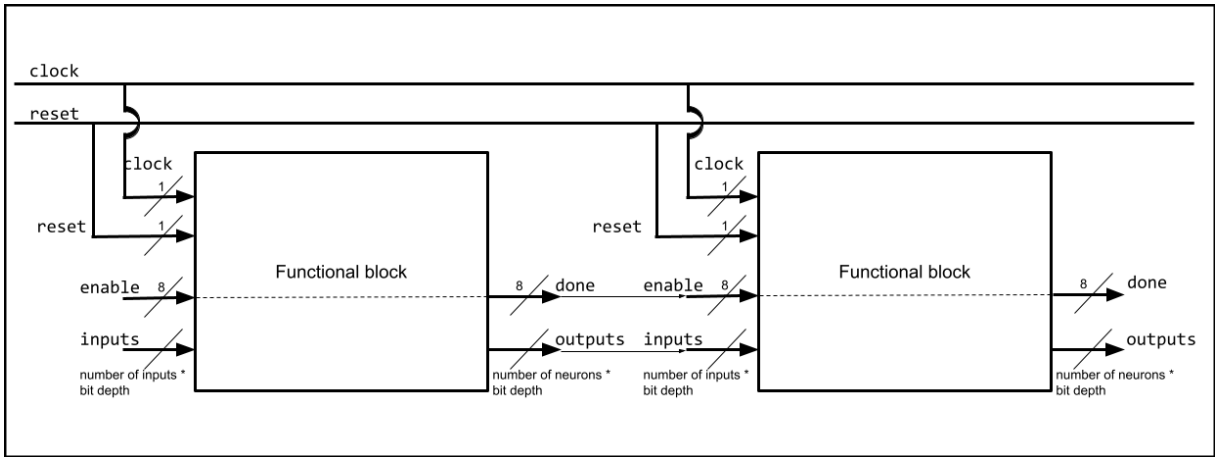


Figure 4 - Functional block synchronization diagram

As shown in figure 4, each hardware block has several input and output signals, however, they always follow this pattern:

- A `clock` signal, for the synchronous execution of the block's operations;
- A binary `reset` input signal, to reset the entire system's state and clear all data;
- An 8 bit pair of input and output `enable` and `done` signals for block synchronization;
- The data `input` and `output` signals, which consist in vectors containing the data to be processed in each block. The data of all inputs or neurons is concatenated in a single vector and must be separated during execution.

To properly synchronize the operations executed by each hardware block, we first use a pair of 8 bit integer signals, `enable` and `done`. The `enable` signal is an input that non-zero values trigger the execution of the operations of the block that receives it. On the other hand, `done` is an output signal that is sent by the block when it is done processing and that the data outputs are ready to be sampled and used by further blocks. When execution is triggered by a non-zero `enable` signal, this value and other inputs are held in memory during execution and its value is outputted as the `done` signal alongside the data outputs. As such, the `enable` value is propagated through

the execution of multiple components alongside its corresponding data and identifies the results, so the neural network input can be matched with its corresponding output.

Another consideration cited is that the `enable` signal and other input data are held in memory locally inside each functional block. This means that once the block's execution is activated by a non-zero `enable`, the previous block doesn't need to hold its output values during this block's execution. Furthermore, this aids pipelining behavior, as the necessary data is stored internally and the execution may be spread along multiple clock cycles, and independent blocks can compute data in parallel. For example, after a block outputs the result corresponding to an input, while the next block processes this result, the first block can start computing the next input.

7.1.2 Input and output data formats

Another consideration to be made in relation to functional block compatibility are the data formats used. They must follow the same rules among different blocks, and the same pattern must be kept when developing new blocks to ensure compatibility.

The pattern, as shown in figure 4, is as follows:

- An input `clk` STD_LOGIC signal;
- An input `reset` STD_LOGIC signal;
- An input `enable` 8 bit STD_LOGIC_VECTOR signal;
- An input `input` STD_LOGIC_VECTOR signal, whose size in bits is the number of inputs of the layer times the bit depth representation used. In this case, using integer-only-quantization arithmetic, the bit depth is 8 bits.
- An output `done` 8 bit STD_LOGIC]_VECTOR signal;
- An output `output` STD_LOGIC_VECTOR signal, whose size in bits is the number of neurons in the layer times the bit depth representation used. For example, in the case of the matrix multiplication, since multiplying two 8 bit values result in a 16 bit value, we use 16 bit depth representation for intermediary calculations to benefit from the temporary increased precision.

7.2 Matrix multiplication block architecture

This functional block follows the input and output pattern described in section 7.1.2. Other than its ports, the differing bit depth representations along execution are worthy of notice. While we use full integer quantization, that is, all the inputs and weights are quantized and have their precision reduced to 8 bits, the bias and outputs have double precision. This is caused by the fact that multiplying two 8 bit values result in a 16 bit value, and the addition by the bias and quantization corrections can profit from the increased precision. Note that the last step in the quantization corrections is casting down back to 8 bit precision, with the proper overflow treatment.

7.3 Rectified linear unit block architecture

Other than following the input and output pattern described in the section 7.1.2, the rectified linear unit consists in a loop that executes a comparison and rectification. The standard rectified linear unit checks if each input is larger than zero, simply outputting the input if it is, and outputting zero otherwise. However, as stated in chapter 5, the neurons use quantized value representation and zero may be represented as a different number depending on the zero-point correction used. Note that the zero-point is a constant, corresponding to the zero-point of the layer's output. As such, this functional block compares the inputs to this threshold instead, defined by a generic constant.

7.4 Quantization correction finite state machine states

The quantization corrections block consists of a finite-state machine of 6 states:

7.4.1 `state_ready`

During the state "Ready", the functional block is idle and awaits a non-zero value in the input `enable` signal, which corresponds to a request from the previous block to process data corresponding to the signal value's identifier.

When the block does receive a non-zero `enable` value, it stores the values on the `inputs` and `enable` signals. This is done as the matrix multiplication block does

not hold the output values stable during further processing; it only guarantees that its output signal is correct on the same clock cycle during which the `done` signal is outputted. As a consequence, the quantization corrections block must store the input values, otherwise they may be lost during the previous block's execution. This is the storage of values that allows for pipelining behavior, that is, the previous block to run in parallel with the quantization corrections block.

Once the values are stored, a change to the next state, `state_zeropoint`, is queued for the next clock cycle.

7.4.2 `state_zeropoint`

During the "Zero-point" stage, the previous block's result is subtracted by $Z_1 a_2^{(k)}$. This is the first state during which the loop unrolling behaviour is present. The previous block's result corresponds to an array, whose elements are the results of each neuron i in a Neural Network layer. As a consequence, there is a $-Z_1 a_2^{(k)}$ correction for each element. Instead of calling a loop and executing every subtraction in the same clock cycle, a signal i is created and incremented every clock cycle, and only the corresponding i^{th} neuron's correction is calculated. Since a single subtraction is calculated every cycle during this stage, the worst-case circuit paths are shorter and faster, speeding up the system's clock frequency.

This also aids the system's scalability: if a neural network layer has many neurons, i can be arbitrarily high and if the loops were not unrolled, the clock frequency would be greatly reduced. In this case, loop unrolling with a high number of neural network layer neurons preserves a high clock frequency by spreading the execution along multiple clock cycles.

Once the signal i's value reaches the number of neurons in the layer minus one, the finite-state machine process queues a change to the next state, `state_M0`, for the next clock cycle. The signal i is also reset to 0, as it is also used in further states' loop unrollings.

7.4.3 `state_M0`

During the "M0" stage, following the efficient handling of the multiplication by M using the normalization of Equation (5.15), the previous stage's result is multiplied by M_0. Since this is a multiplication by a higher precision fixed-point value, a resource intensive operation, it has its own dedicated stage. Note that similarly to the last step, this operation is also done individually for each neuron's result, it also corresponds to a loop, which is also unrolled to preserve clock frequency. The loop unrolling is also programmed iterating the signal i,

To properly compute the multiplication by a fixed-point number, a VHDL-2008 package, the `IEEE.FIXED_PKG`, is included and used during this stage. This implementation includes efficient algorithms for operations with fixed-point numbers and convenient variable types to represent such numbers. Namely, M_0 is stored as a `ufixed(-1 downto -32)` signal, which represents a unsigned fixed-point number of 0 bits for the decimal part and 32 bits for the fractional part. This choice of bit representation corresponds to the M_0 definition which states that the parameter is in the interval [0.5, 1), as such there is no need for bits to represent the decimal part of the number.

When all the multiplications by M_0 are done, indicated again by the signal i's value reaching the number of neurons minus one, the finite-state machine queues the next state, `state_bitshift`, for the next clock cycle, and resets the signal i to 0.

7.4.4 `state_bitshift`

During the "Bitshift" state, the second step of the efficient handling of the multiplication by M following Equation (5.15) is executed.

Since in Equation (5.15) n is always a positive integer, the exponent is always negative, corresponding to a right shift by n bits.

As specified in (JACOB et al., 2017) and described in the 5 chapter, the bit-shift must have correct round-to-nearest behavior. This behavior must specifically round away from zero, while the shift right instruction in VHDL truncates the result and corresponds to a biased round towards minus infinity. This biased rounding behavior was observed to cause significant loss of accuracy in neural network inference (JACOB et

al., 2017), and must be avoided.

The hardware implementation to achieve rounding behavior away from zero first adds a rounding parameter equal to $2^{(n-1)}$, and only then executes the necessary shift right correction corresponding to the multiplication by 2^{-n} defined in Equation (5.15) to the previous state's result.

When the bitshifts are finished, the finite-state machine queues a state change to the `state_overflow` state

7.4.5 `state_overflow`

During the "Overflow" state, the results of the previous state are type cast towards signals of the desired precision. During the matrix multiplication and bias sum, the 8 bit signal was transformed into a 16 bit signal, and this extra precision was carried over the quantization corrections. The layer outputs must be represented by 8 bit signals to preserve the quantization benefits and so they can be used themselves as another layer's inputs.

This stage checks whether the results' values overflow or underflow the 8 bit maximum and minimum values, truncates them as needed and cast them towards an output storage signal. This output storage doesn't yet correspond to the functional block output.

Note that this must be done to every value in the quantification correction results array, and consists in a loop, which is similarly unrolled.

Once the state finishes the type casting towards the output storage signal, the finite-state machine queues a state change to `state_end`.

7.4.6 `state_end`

During the state "End", the output signal is outputted as the functional block output, accompanied by a `done` signal corresponding to the `enable` value stored during the `state_ready` state. The change in `done` signal with a non-zero value marks that the output is ready to be used in another layer or network output, and the `done` value itself identifies the result.

No correction operations are executed during this state.

After outputting the results, the finite-state machine queues a state change to `state_ready`, and the block is ready to receive another set of inputs.

8 TESTS

To evaluate the system's operation, a test that covers all the steps in generating a hardware accelerator is necessary. As such, the following was decided:

1. An artificial neural network, compatible with the system, is to be designed and trained;
2. The neural network is to be used as input in the generator system;
3. The generated hardware must be tested and evaluated.

First, since the chosen hardware accelerator architecture paradigm was dedicated circuit design, each activation function and network topology must be programmed separately in its own hardware functional block, which is a time consuming task. As a consequence, the resulting proof of concept system is only compatible with the Rectified Linear Unit (ReLU) function and the fully-connected network topology, chosen by their popularity and simplicity. Usual popular neural network models available use several functions and topologies, as they are at disposal in the software context, with software libraries being easily imported and used. However, to test the functionality of accelerator generation and performance, a neural network has to be programmed and trained using only fully-connected layers and the ReLU function.

To facilitate testing, a neural network was implemented and trained to execute the Exclusive OR (XOR) function. As a consequence, the intermediary data can be checked manually more easily, and its linearly inseparable nature was important in neural network and machine learning history (HORNIK; STINCHCOMBE; WHITE, 1989). Figure 5 demonstrates the linear inseparability of the XOR function. The colours of the points indicate the output values of the functions, while the axis represent the inputs. The blue points represent the $\{0, 0\}$ and $\{1, 1\}$ inputs, resulting in 0, and the yellow points represent the $\{1, 0\}$ and $\{0, 1\}$ inputs, resulting in 1. As the dotted lines show, there is no line on the plane that can be drawn with all blue points on one side, and all yellow points on the other, and as such, XOR is a linearly inseparable function. An example of a linearly separable function is the AND function, shown in figure 6. The

dashed line shows a possible line that can separate the sets of points related to inputs resulting in each output. The linear inseparability means that single-layer networks, such as perceptrons, cannot achieve the non-linear behavior of its output. As such, implementing an XOR function demonstrates that the neural network hardware accelerator generated by our system is capable of synthesizing deep learning networks with hidden layers.

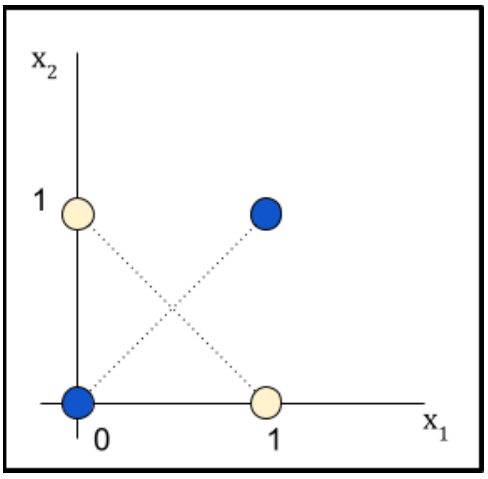


Figure 5 - Graph showing the linearly inseparable nature of the Exclusive OR function

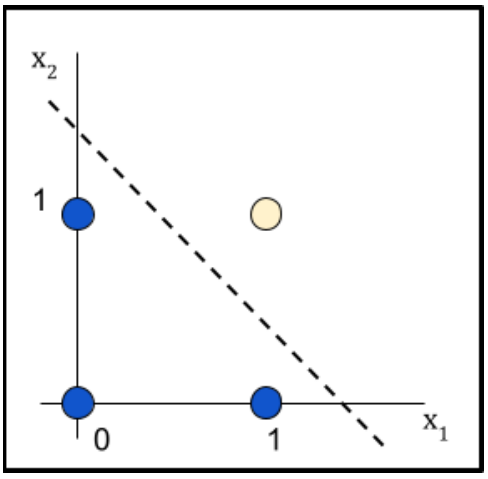


Figure 6 - Graph showing the linearly separable nature of the AND function

To implement the Exclusive OR function, a neural network with the topology shown in figure 7. This figure shows that the network receives two inputs, represented by the two input layer nodes; the network outputs a single value, represented by the output layer node; there is a single hidden layer, with 3 neurons; and both the hidden and output layers use a bias, represented by the gray nodes. All of the neurons in the hidden and output layer execute the Rectified Linear Unit activation function, as compatibility

with the hardware accelerator generator system demands. Furthermore, as shown by the arrows connecting each node, the neuronal interconnectivity rules used by all the layers is the fully connected layer, with every neuron of a layer outputting values to every neuron of the following layer, also according to the generator's compatibility.

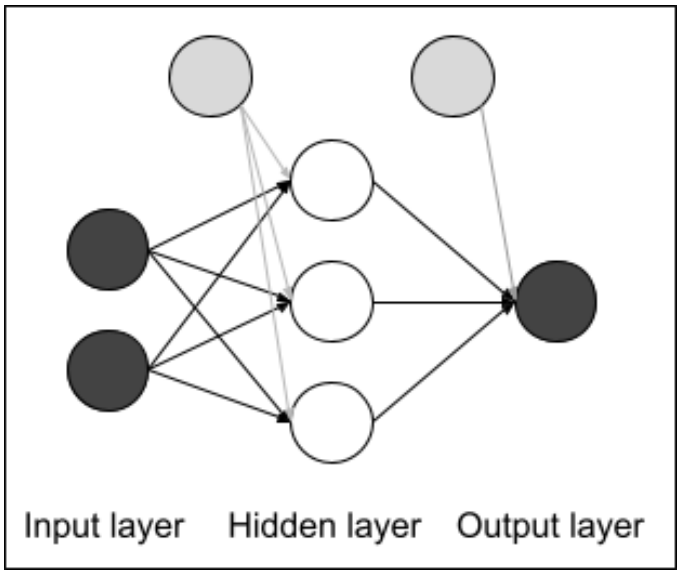


Figure 7 - Graph showing the topology of the implemented neural network that executes the Exclusive OR function

Additionally, the neural network used in the prototype described in Chapter 4 which implements a classifier over the Iris dataset was also used to verify the generation behavior.

As shown in figure 8, the neural network consists in 4 inputs, 4 hidden layers, and 3 outputs. Note that only the first hidden layer, with 3 neurons, uses a bias, while all the other layers do not, and have 5 neurons each. Another important consideration is the use of one-hot encoding, as described in the Chapter 4. This representation means that while the classifier has 3 possible output classes, the dataset was configured and the network trained considering 3 binary outputs, one for each possible classification result, instead of a single output whose value determines the class. This method increases classification precision and distinction between the different results, and the value obtained can be used as a measure of certainty of the classifier in each inference result.

Also note that the grayed-out neuron connections in figure 8 are merely an attempt in visual clarity, as every neuron connection is shown to highlight the fully-connected

layer interconnectivity.

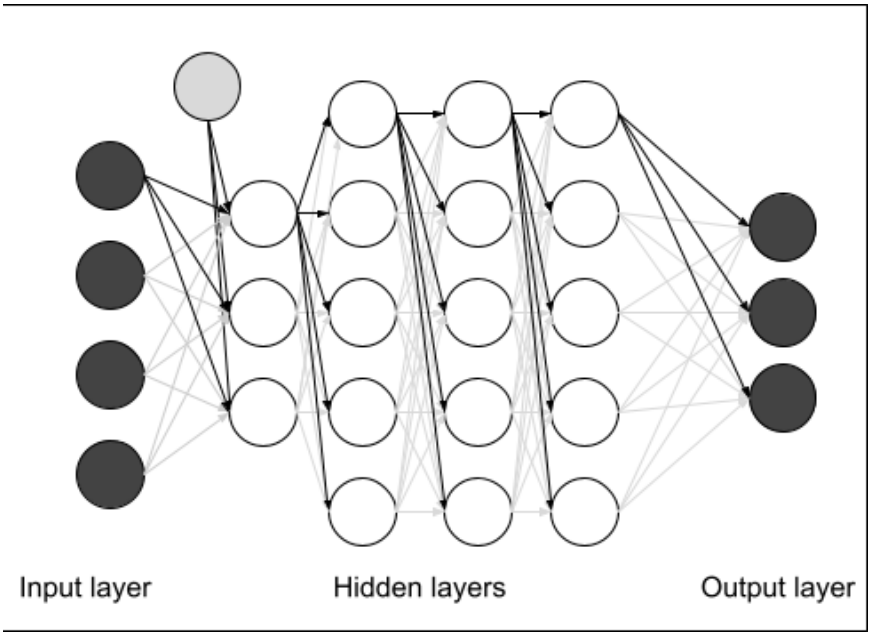


Figure 8 - Graph showing the topology of the implemented neural network that executes a classifier over the Iris dataset

Both of these networks are programmed using Tensorflow Lite's framework, with the Keras machine learning library, and the resulting models are exported using the corresponding libraries' export functions.

The models were then imported into the generator system, which ran with no issues and wrote files corresponding to the hardware accelerators that implement these networks.

The generated hardware was tested by simulations using the Intel's ModelSim software in order to verify whether the behavior of the hardware matches the original neural networks in software. Note that the expected result is the exact same as the neural network implemented in software. This is due to the system mirroring the same algorithms and execution of the neural network, including the same quantization representations, quantization steps and corrections necessary. During simulation, we also obtain the number of clock cycles until the first result is outputted, and the cycles per result once the system's pipeline stabilizes.

Then, the accelerators were compiled and synthesized using Intel Quartus Prime, using the Altera Cyclone® IV EP4CE22F17C6N FPGA used by the DE0-Nano Terasic board as the target. This device was chosen as it is relatively cheap, available

and specifically designed for low-power and small applications. The compilation and synthesis in Intel Quartus Prime offers several results and reports, as well as tools for further analysis of the generated system.

As previously mentioned in the Chapter 3.2, it is not sufficient to simply evaluate the execution speed of neural network inference results to evaluate the hardware. This is due to the nature of the target environment of the hardware accelerators, edge computing in the Internet of Things. This environment often restricts energy availability and device sizes, and as such, an otherwise fast and successful accelerator may consume too much energy or not fit in specific devices for certain applications. As a consequence, more data needs to be obtained during testing.

The information obtained from this execution were the following:

- Maximum system clock frequency achieved;
- Cyclone IV FPGA implementation integrated circuit demand, such as logic elements, registers and multipliers needed and the amount available in this device;
- Using the Power Analyzer tools, we can simulate energy expenditure during usual use.

By combining the number of clock cycles until the first result outputted and the number of cycles per result once the system's pipeline stabilizes, and the maximum system clock frequency achieved by the design compiled, we can calculate the latency in seconds for these results.

Additionally, we can compare the Cyclone IV FPGA implementation integrated circuit demand with the amount available in the device, and evaluate if the design fits a compatible FPGA. By evaluating the total number of logic elements of the Lattice Semiconductor's iCE40UP5K-UWG30 and comparing it with the current device choice, the Altera EP4CE22F17C6N Cyclone IV, we concluded that while the former has an especially low number of components and ultra-low-power consumption, these are unnecessarily low and significantly restrictive to usual neural network accelerators following our architecture. As such, by choosing the Cyclone IV FPGA, which is also suitable for Internet of Things applications, but is less restrictive, we can maintain the relevancy

and real life applicability of the tests executed while easing hardware restrictions on the designs generated.

Then, we can evaluate the results obtained by the Intel's Quartus Power Analyser tools and compare with reasonable energy harvesting devices, such as mini solar panels suitable for an Internet of Things application. With this comparison, we can assess the compatibility of the resulting neural network hardware accelerator with the restrictive energy availability in edge computing contexts.

Once these results are obtained, they were compared to the neural network's inference in software, using the original model's implementation. The application was executed using a personal computer Intel i7-8700k CPU, which was chosen for availability and as it is a good example of a high-mid range full scale processor. This comparison was chosen to highlight the gainst in performance with the implementation of the dedicated circuits used by the design methodology chosen for the generator system. Additionally, it will help to evaluate the order of magnitude of power expenditure necessary for both of the hardware choices, and highlight the accelerator's compatibility with the restrictive Internet of Things environment.

To test the software's performance in inference execution using the personal computer processor, a Python script was created to measure execution time. The script begins by the loading the artificial neural network model into an Tensorflow Lite Interpreter object. The interpreter is the main interface for running Tensorflow Lite models, and include several methods for setting and reading tensors and configuring runs. Then, once the model is imported and allocating the memory for the model's tensors, different compatible input arrays are created and stored. A timer then is started, using the (perf_counter_ns()) function from Python's time module. The reasons to use this specific function as the timer are that the performance counter has the highest available resolution for measurement, includes time elapsed during sleep, is system-wide, and avoids the precision loss caused by the `float` type as it counts in nanoseconds. Next, a series of batches of inferences are run. Each inference consists in a call to the `set_tensor()` method of the Tensorflow Lite Interpreter interface, which is configured to set the input array of the neural network model, and a call to the `invoke()` Interpreter method, that triggers the execution of the neural network. Each batch of inferences consists in repetitions of this pair of input setting and invoking of inferences.

Several batches of inferences are made to attempt to avoid timing issues related to the computer architecture of the hardware and programming language used. Furthermore, the entire program is ran twice, and the timing results recorded are those of the second run. This is done to reduce latencies related to memory caching, and once ran, the relevant data are already stored in faster memory formats. This is evidenced by the fact that the second execution runs are significantly faster than the first, without changing any other test parameters.

One possible issue, for example, is the start of an inference batch with a cold cache, that is, when the cache memory, the fastest and closest memory to the CPU processing, is empty or has irrelevant data, requiring a slower read from the main memory for execution. Once the execution of many steps is done, the cache may speed up after a cold start, and as such the latency for each inference in batches with a larger number of calls speeds up. By running several batches with different number of inferences each, we try to isolate these issues and evaluate the neural network itself.

The personal computer's CPU energy expenditure, on the other hand, was simply determined by using the manufacturer's divulged thermal design power during maximum theoretical load. Note that by using such estimates, we can only ascertain conclusions arising from the order of magnitude of the values obtained. However, given the nature of the comparison between a full-scale personal computer's CPU, unsuited for the Internet of Things edge computing environment, and a compatible, small scale, low power device with an FPGA, such conclusions are significant and adequate.

9 MAIN RESULTS

We executed the tests following the methodology described in Chapter 8 using the neural networks generated for the Exclusive OR and Iris classification datasets, with the following results.

9.1 Exclusive OR test results

9.1.1 VHDL logical correctness

Following the test methodology described in Chapter 8, the generated VHDL code's behavior was first simulated using Intel's Modelsim software. Using this software, the logical behavior of the hardware described by the VHDL language is simulated, including a stimuli file that generates sample inputs to the system, and reads the outputs generated. As such, we can evaluate the correctness of the neural network algorithm implemented by the VHDL by comparing the resulting outputs with those expected, which are the results generated by the original neural network when executed in software.

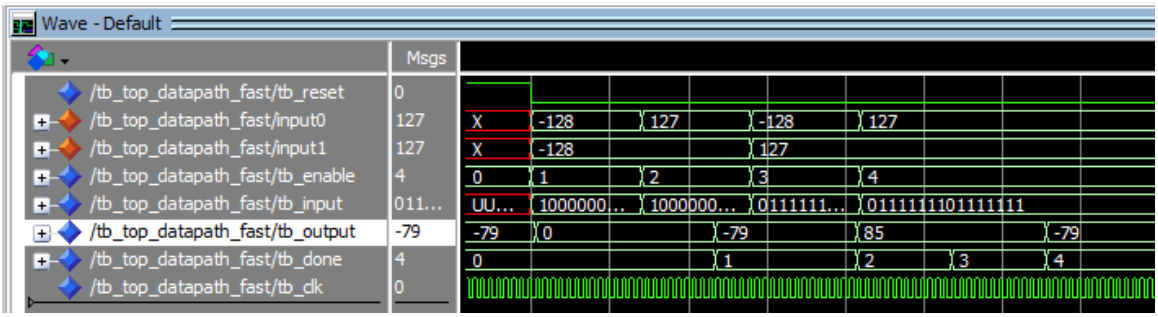


Figure 9 - Exclusive OR logical correctness Modelsim simulation waveform results

In Figure 9, the simulated Exclusive OR hardware's inputs and outputs are shown in a waveform format. Each row corresponds to a signal being tracked by the simulation, with the text on the left corresponding to the signal's name and address inside the hardware. For example, the signal labeled "`/tb_top_datapath_fast/tb_input`" corresponds to a signal called "`tb_input`" located within the hardware entity "`tb_top_datapath_fast`". The "`tb`" prefix represents a "testbench" entity, connected

to the top hierarchical hardware entity "top_datapath", used only in tests in order to generate the simulation's inputs and receive the system's outputs. Note that the hardware organization was described in Chapter 7. Then, the waves to the right of the labels correspond to the signal's values along the time of the simulation. For example, we can see that the "/tb_top_datapath_fast/tb_enable" signal takes the following values, in this order: 0, 1, 2, 3, 4.

The signals represented in this simulation are as follows:

- tb_reset, which corresponds to the reset signal;
- tb_input, which corresponds to the input data given to the neural network. In this case, the Exclusive OR network receives two concatenated quantized 8 bit variables corresponding to the two logic gate inputs;
  - Note that the two 8 bit variables are separated into the signals input0 and input1 for visual clarity.
- tb_enable, which corresponds to the enable signal, that triggers the neural network execution and labels each input with a numerical value;
- tb_output, which corresponds to the output signal, that corresponds to the neural network's quantized 8 bit output;
- tb_done, which corresponds to the done signal, that identifies the readiness of the outputs, as well as identifying them, matching them with the enable values paired with the respective inputs;
- tb_clk, which corresponds to the system's clock signal.

From this simulation's results in Figure 9, we can first analyse the input values. In order, the network receives the inputs $[-128, -128], [127, -128], [-128, 127], [127, 127]$, respectively accompanied by the enable values 1, 2, 3, 4. Since the network has been quantized into 8 bit representation precision, the input pairs correspond to the values $[0, 0], [1, 0], [0, 1], [1, 1]$. The neural network implements the Exclusive OR function, and as such, the expected output values for the outputs labeled by the done signal's values 1, 2, 3, 4 are $[0], [1], [1], [0]$. The outputs labeled by done values 1, 2, 3, 4 are

$[-79], [85], [85], [-79]$, that when dequantized, correspond to the expected output values $[0], [1], [1], [0]$.

The input and output quantization rules were automatically determined by the neural network training, and are the following:

$$input : real\_value = 0.0039215 * (quantized\_value + 128) \tag{9.1}$$

$$output : real\_value = 0.0060795 * (quantized\_value + 79) \tag{9.2}$$

For example, $(127 + 128) * 0.0039215 = 0.9999825$ and $(-128 + 128) * 0.0039215 = 0$, approximately corresponding to the expected binary $0, 1$ input values.

In conclusion, the neural network successfully receives quantized inputs following the quantization rules set by the network in software, labeled by `enable` signal values; the network then processes these inputs and outputs the expected results, using the correct quantization rules, and matches the outputs with the `enable` values outputting them as the `done` signal.

9.1.2 VHDL implementation parameters for Cyclone IV FPGA

As described in Chapter 8, the generated VHDL code was compiled and synthesized using Intel's Quartus Prime software, which outputs several reports on different parameters and results obtained during processing. One of these reports, the mapping and fitter results, contain data related to the number of total logic elements, registers, pins and multipliers requested by the current design to be synthesized in the compilation's configured targeted hardware. In this case, we targeted the Altera's Cyclone IV EP4CE22F17C6N FPGA, and the results and percentages are related to the available integrated circuits in this device. This is an important result, as if the design demands more components than there are available in the device, the accelerator would not be synthesizable in this specific hardware. The implementation parameters, supply and demand of logic components is shown in Table 1.

The data in Table 1 shows that the design fits successfully into the target FPGA, as well as the report indicating that the fitter status was successful, and thus, is synthesizable.

Table 1 - Exclusive OR Quartus Prime post-fitting hardware component report for Cyclone IV FPGA target

Components	Demand/Availability
Logic Elements	896/22,320 (4%)[a]
Registers	483
Pins	42/154 (27%)[a]
Multipliers	8/132(6%)[a]

[a]Percentages correspond to hardware available in the Altera Cyclone IV EP4CE22F17C6N FPGA

Furthermore, the compilation results in a timing report, which indicates the maximum clock frequency obtained with which the required design logic can be successfully executed when ran using the target FPGA. This clock frequency is shown in Table 2, and this result is used in further tests to evaluate the hardware's execution speed.

Table 2 - Exclusive OR Quartus Prime timing maximum clock frequency report for Cyclone IV FPGA target

Parameter	
Maximum clock frequency	100.61 MHz

9.1.3 Execution speed tests

As explained in Chapter 8, there are two different execution speed parameters to be determined during this test, due to the use of pipelining behavior in the hardware architecture: the latency measured between the first input and until the first result is outputted; and the time taken between each subsequent result output once the system has stabilized. Since the hardware is pipelined, that is, with multiple serial components able to simultaneously and independently execute data pertaining to different inputs, when the first inference is requested, the circuit is idle. However, during the execution of this first input, other data is inputted and the pipelining behavior starts increasing execution efficiency and speed. As such, there is a higher latency until the first input's results are done, but once the results begin flowing, the time taken between each result is lower than the initial latency, resulting in an increase in throughput.

To obtain both of these time measurements for the generated hardware accelerator, we first simulate the circuit's behavior in a Intel's Modelsim logical simulation. This

software will simulate inputs and the respective logic outputs with precision, however the system clock generated for the simulation uses an artificially set frequency. We can obtain the number of clock cycles necessary for both of the required timing results, and by replacing the artificial clock frequency by the one obtained by the Intel Quartus' design compilation timing results, we can obtain the time results in seconds. The results in number of clock cycles are shown in Table 3. Furthermore, the timing report resulting from the design's compilation in Quartus Prime, concluding the FPGA's maximum clock frequency for this hardware's execution, is shown in Table 3.

By the definition of period, shown in Equation 9.3, we can obtain the system's clock period from the obtained maximum frequency results. Then, by multiplying the number of clock cycles by the obtained period, we obtain the timing parameters in seconds.

$$period = \frac{1}{frequency} \tag{9.3}$$

Table 3 - Exclusive OR hardware accelerator timing parameters results

Parameters	
Clock delay until first result	23 Clock cycles
Stable clock delay per result	12 Clock cycles
Maximum clock frequency	100.61 MHz
Clock period	9.93 ns
Time until first result	228.60 ns
Stable time per result	119.27 ns

Then, also following the methodology described in Chapter 8, we tested the time taken to execute the same Exclusive OR neural network inferences in software, using a full-scale personal computer Intel i7-8700k CPU. Multiple runs with different numbers of inferences were executed to help isolate interfering factors, such as pipelining behavior and cold caches. The results obtained are shown in Table 4

Note that Table 4 and Figure 10 show that the CPU optimizes its running times and takes significantly longer to run low number of inferences, as there are delays related to starting the operations and optimizations in running large batches. Figure 10 shows that in execution requests where the number of inferences requests was smaller, the time elapsed per inference is longer, and vice versa. An example of an optimization that CPUs employ is speedup related to data caching of relevant processing data so

Table 4 - Exclusive OR software inference timing results for the personal computer CPU

Number of inferences in run	Time taken to run	Time per inference
1	83600 ns	83600 ns
2	94200 ns	41800 ns
4	137600 ns	20900 ns
12	221700 ns	6966.7ns
36	322700 ns	2322.2ns
108	806300 ns	774.1 ns

that further requests access data stored in memory with faster access times. When a batch has a low number of inferences, the memory cache, the fastest memory in a CPU, is empty or has irrelevant data, forcing the CPU to read from slower memories. In the case of an execution batch with a high number of inferences, the cache memory can be loaded more efficiently with relevant data, and the computations can read data from a faster memory, resulting in faster times per inference.

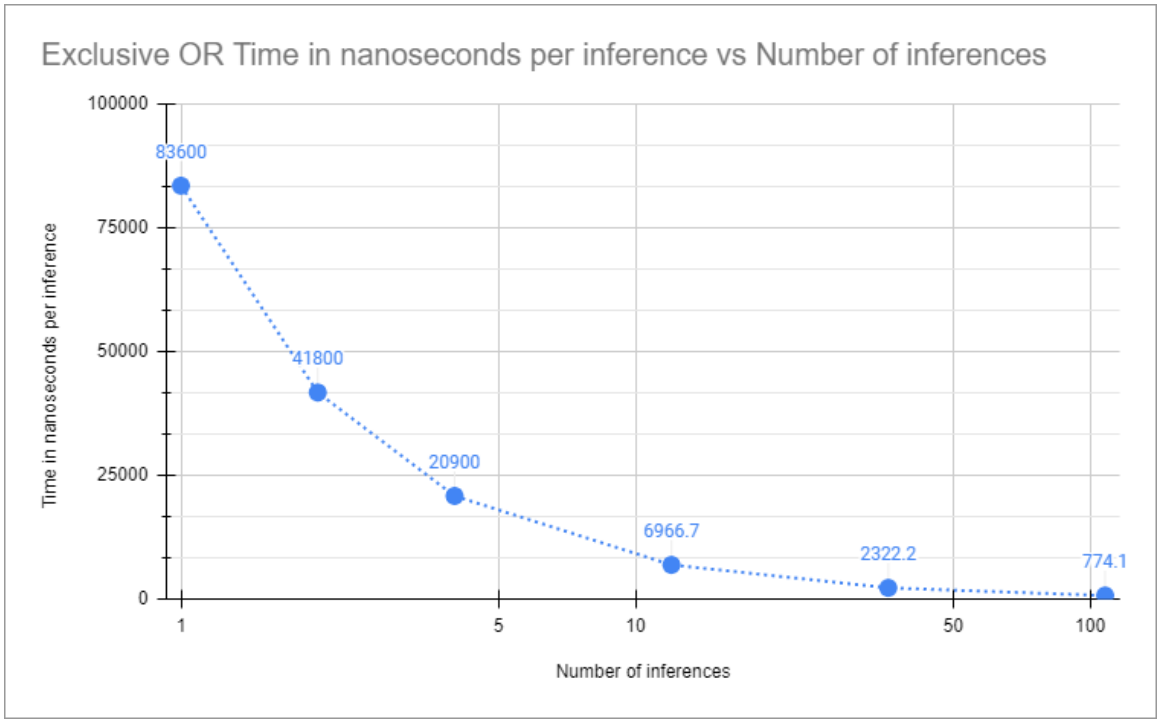


Figure 10 - Exclusive OR time per inference in software, ran using Intel i7-8700k CPU

By comparing the data obtained in both tests, we can evaluate the performance of the obtained hardware accelerator in optimizing the throughput of neural network inference results. This comparison is shown in Table 5.

As shown in Table 5, there are significant gains in latency to first result when com-

Table 5 - Exclusive OR timing results comparison between hardware accelerator and personal computer CPU

	Hardware	
Time delay	***FPGA Cyclone IV***	***Intel i7-8700k PC CPU***
Time until first result	228.60 ns	83600 ns
Stable time per result	119.27 ns	774.1 ns

paring the generated neural network accelerator to the personal computer Intel i7-8700k CPU. Additionally, when stabilized, the time per result throughput is still faster in the FPGA.

9.1.4 Energy expenditure tests

As explained in Chapter 8, the energy expenditure results for the synthesized hardware accelerator were taken from the Power Analyzer tool from the Intel Quartus Prime software. This tool was used once the hardware design was compiled, and it estimates the power consumption of the FPGA device. Note that while this is an estimate, it is precise enough for considerations over the efficiency gains when compared with a personal computer CPU, and by evaluating its order of magnitude we can reasonably draw conclusions as to the final hardware's compatibility with the Internet of Things edge computing energy availability.

Then, the result obtained in the Power Analyzer tool report is compared with the CPU Thermal Design Power divulged by the personal computer CPU manufacturer, Intel, to draw a comparison as to the order of magnitude of energy expended by both hardware solutions. This comparison is shown in Table 6.

Table 6 - Exclusive OR energy expenditure comparison between FPGA accelerator and personal computer CPU

	Hardware	
	FPGA Cyclone IV	***Intel i7-8700k PC CPU***
Total thermal power dissipation	126.45 mW	95 W

As shown in Table 6, there are significant energy efficiency gains while using the generated neural network hardware accelerator when compared with a full-scale personal computer Intel i7-8700k CPU.

9.2 Iris classifier test results

The tests conducted using the Exclusive OR neural network, while significant in demonstrating the basic functionality of each of the steps of the entire system, doesn't represent a realistic application, and therefore, the conclusions drawn from these results have their limitations in reach. As such, to help draw more significant conclusions, this more complex problem, consisting of a more complicated and realistic dataset as well as a bigger neural network, was tackled.

The test methodology used is the same as the Exclusive OR example, and as such the results can be compared with greater significance, and rough conclusions pertaining to the order of magnitude of the scalability of the system can be drawn. For example, since the neural network to be used to generate the hardware accelerator has more layers and neurons, it is interesting to observe what sort of consequences in energy expenditure, FPGA component demands and execution speed this causes.

Similarly to the tests related to the Exclusive OR example, the neural network was inputted into the generator system, which was ran and generated VHDL files corresponding to the hardware accelerator. These files were used in a Modelsim logical simulation, compiled and synthesized using Quartus Prime software, which outputted several reports.

9.2.1 VHDL logical correctness

Following the same methodology as the simulation done for the Exclusive OR example, we tested the logical correctness of the generated VHDL code for the Iris dataset classifier. The resulting Modelsim waveform results are shown in Figure 11.

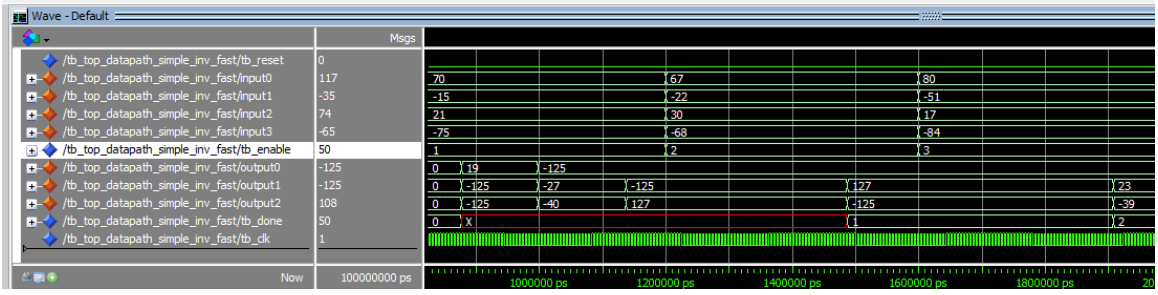


Figure 11 - Exclusive OR logical correctness Modelsim simulation waveform results

As the Exclusive OR network results, this waveform also contains signals corresponding to the inputs, outputs, `enable` and `done` signals, as well as the system clock and a `reset` signal, with their rows and waveforms labeled by the text on the left.

However, since the test data includes 50 classifier samples, only the first two outputs are shown in Figure 11. The rest of the simulation results are shown in the Appendix A. In this network, the input and output quantization rules are as follows:

$$input : real\_value = 0.0301960 * (quantized\_value + 128) \tag{9.4}$$

$$output : real\_value = 0.0043238 * (quantized\_value + 125) \tag{9.5}$$

For example, the first input values $[70, -15, 21, -75]$, labeled by the `enable` value 1, correspond to the floating point values $[6, 3.4, 4.5, 1.6]$, which correspond to the first inputs in the Iris test samples dataset, shown in the Appendix A. As an output dequantization example, on the other hand, could be the outputs [-125, 127, -125], labeled by the `done` value 1, correspond to the values $[0, 1, 0]$. This value represents a classification in the second class. All the values were dequantized and compared with the expected test dataset classification labels, obtained by running the original neural network in software using the same inputs, and the results were correct as expected. Note that there were small inconsistencies, possibly arising from rounding and truncating errors during the manual input and output quantization, but that didn't affect the system's overall classifying performance. The entire result comparison is shown in the Appendix A.

In conclusion, the neural network hardware accelerator successfully receives the quantized inputs and outputs the expected results.

9.2.2 VHDL implementation parameters for Cyclone IV FPGA

As the Exclusive OR example, the generated VHDL files were compiled and processed by Quartus Prime, which generated a post-fitting analysis of the components necessary to synthesize the design. These results are shown in Table 7.

Table 7 - Iris classifier Quartus Prime post-fitting hardware component report for Cyclone IV FPGA target

Components	Demand/Availability
Logic Elements	7,117/22,320 (32%)[a]
Registers	2304
Pins	74/154 (48%)[a]
Multipliers	20/132(15%)[a]

[a]Percentages correspond to hardware available in the Altera Cyclone IV EP4CE22F17C6N FPGA

The FPGA component demand and availability results of Table 7, as well as the report indicating that the fitter status was successful, show that the design fits successfully in the target Cyclone IV EP4CE22F17C6N FPGA, and thus is synthesizable.

Furthermore, when compiling the VHDL files, Quartus Prime also outputs a timing report that shows the maximum clock frequency with which the target hardware can execute the design with the expected behavior. This maximum clock frequency is shown in Table 8, and this value is used in execution speed tests.

Table 8 - Iris classifier Quartus Prime timing maximum clock frequency report for Cyclone IV FPGA target

Parameter	
Maximum clock frequency	18.35 MHz

9.2.3 Execution speed tests

The Iris classifier execution speed tests were conducted with the same methodology as those of the Exclusive or example. As such, the VHDL code is simulated using Intel's Modelsim software and the number of clock cycles are counted for two separate events: the latency between the first data input in the network and when its result is outputted; and the number of clock cycles per result once the system is stabilized. These results are shown in Table 9.

Then, following the methodology described in Chapter 8, we tested the inference time in multiple runs in software with the personal computer Intel i7-8700k CPU, with different numbers of inferences, done twice each and the fastest result recorded to isolate interferences. The results are shown in table 10, and the graph in Figure 12

Table 9 - Iris classifier hardware accelerator timing parameters results

Parameters	
Clock delay until first result	137 Clock cycles
Stable clock delay per result	80 Clock cycles
Maximum clock frequency	18.35 MHz
Clock period	54.49 ns
Time until first result	7 465.94 ns
Stable time per result	4 359.67 ns

highlights the effect of software optimizations and pipelining, speeding up time per in bigger batches of inferences.

Furthermore, we can compare these results with those obtained during the Exclusive OR tests. The bigger neural network's computations are spread along more hardware components, as shown in Table 7, the loop unrolling techniques and higher number of sequential layers result in more clock cycles until the results are ready, and the increased neural network complexity results in a lower maximum clock frequency.

Table 10 - Iris classifier software inference timing results for the personal computer CPU

Number of inferences in run	Time taken to run	Time per inference
1	151 600 ns	151 600 ns
2	156 000 ns	78 000 ns
4	166 000 ns	41 500 ns
8	201 800 ns	25 225 ns
16	269 900 ns	16 868 ns
32	405 000 ns	12 656 ns
50	552 300 ns	11 046 ns

The results obtained during tests on the personal computer Intel i7-8700k CPU are then compared with the final hardware accelerator execution time results in Table 11.

Table 11 - Iris classifier timing results comparison between hardware accelerator and personal computer CPU

	Hardware	
Time delay	***FPGA Cyclone IV***	***Intel i7-8700k PC CPU***
Time until first result	7 465.94 ns	151 600 ns
Stable time per result	4 359.67 ns	11 046 ns

Similarly to the results obtained with the Exclusive OR example, Table 11's results

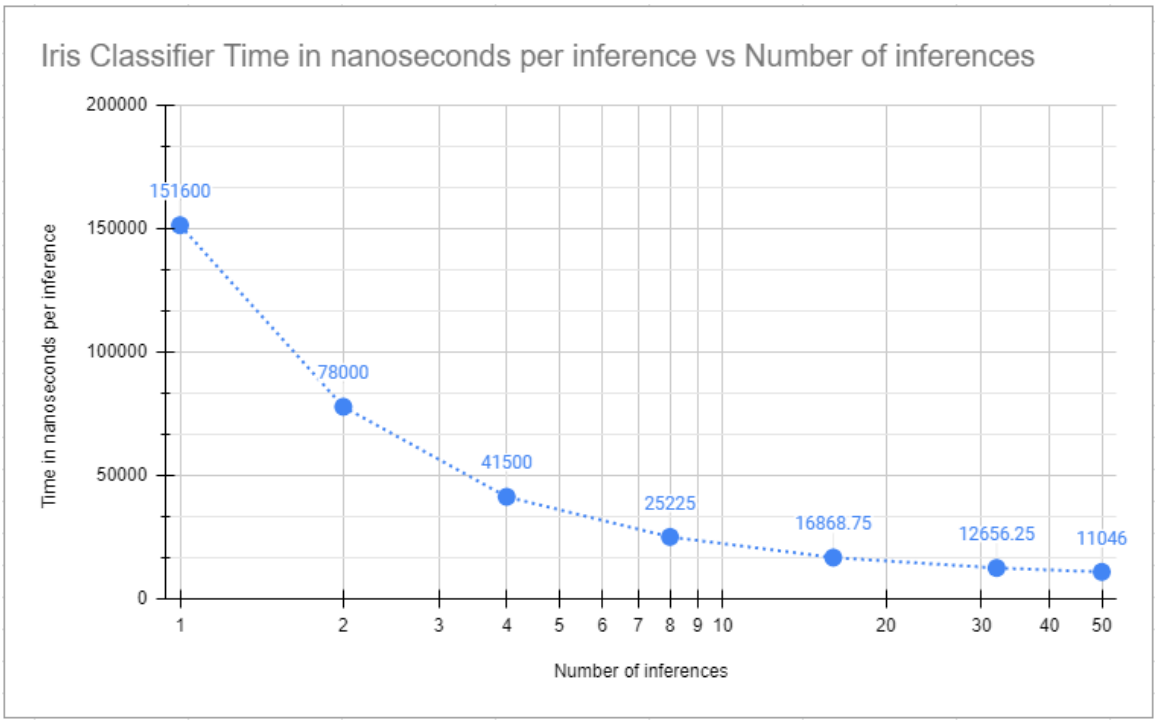


Figure 12 - Iris classifier time per inference in software, ran using Intel i7-8700k CPU

indicate that the low-power FPGA hardware accelerator has significant latency until first result and stabilized result throughput timing gains over the personal computer CPU.

9.2.4 Energy expenditure tests

Following the same methodology used in the previous energy expenditure test, we ran the Intel Quartus Prime Power Analyzer tool, and extracted the estimated energy expenditure data from the resulting report. This result is shown in Table 12, and also compared with the personal computer CPU Thermal Design Power divulged by Intel.

Table 12 - Iris classifier energy expenditure comparison between FPGA accelerator and personal computer CPU

	Hardware	
	FPGA Cyclone IV	***Intel i7-8700k PC CPU***
Total thermal power dissipation	121.72 mW	95 W

The order of magnitude of the energy expenditure results from the Power Analyzer report displayed in Table 12 is comparable with that of the Exclusive OR example results, and both express a significant gain in energy efficiency for the execution of the neural network in comparison with the personal computer CPU.

10 CONCLUSIONS

10.1 Threats to validity

Firstly, in order to properly evaluate the results of this work, we must assess the limitations of its results. The first of the threats to the validity of this project is the narrow breadth of compatible neural network topologies and activation functions. This is a consequence of the dedicated circuit design paradigm chosen, as each of the functions and topologies must be implemented directly in hardware. With the overlay-based co-design methodology, the other main design paradigm, there would be a larger overlap between the already-existent software implementations of these functions and how they would be implemented in the hardware accelerator. The reason is that this design implements processor cores that are programmed with instructions to execute the neural networks, and the programming is more closely related to software. While easier, the difficulty to implement a large library would remain, as the state of the art of neural network and machine learning design uses a vast quantity of different functions and layers.

As a consequence, since the system is in such a state with this proof of concept, its utility and readiness to be used by end users as it is is limited. With a limited usability, there is less incentive for the system to be expanded and to increase its popularity. To properly address the objectives numbered in Chapter 3.2, the system must be attractive to be expanded by the open source programming community, and to ensure its longevity and usefulness. Consequently, more activation functions and popular layer and network topologies must be implemented in order to encourage the system's use, and its current state is limited.

However, care has been taken in the system's design, whether it be during the generation steps of the actual generated hardware architecture to facilitate its expansion. The modularity of the functional blocks used by the dedicated circuit design, as well as the design rules and patterns pertaining to execution, synchronization and data formats will help efforts to expand the system and ease this threat to the validity of the project.

Furthermore, since the range of functions implemented is limited, the tests performed to evaluate the operation of this system are less significant and noteworthy. Ideally, to reinforce the system's value and usability, a popular neural network model such as the YOLO, You Only Look Once classifier (REDMON et al., 2015), should be implemented and a hardware accelerator, generated. Then, a recognizable model, with important consequences would represent a significant result for the generator system.

Even so, the YOLO classifier uses convolutional layers, which represent another sort of neural network topology, with different neuronal interconnectivity rules, as well as max pooling layers, that require a different kind of activation function. Although these functional blocks are not yet programmed and the lack of significant results such as the implementation of a famous model like YOLO, note that once these blocks are implemented, a hardware accelerator for the YOLO classifier could be generated by the system, highlighting its flexibility once the functional library is expanded.

Another consideration to be made in respect to the tests executed is that the comparisons of energy expenditure and execution speed were made between a small device with an low-power FPGA and a full-scale high-mid range personal computer processor. This comparison is noteworthy and relevant to highlight the gains in performance and energy efficiency, and help characterize and establish the generated hardware as an adequate hardware accelerator to be used in the context of edge computing for Internet of Things devices. This is due to the high restriction nature of the Internet of Things' environment, that may restrict energy availability and device size. However, an interesting comparison would be to showcase the hardware accelerator generated by this system against similar accelerators, such as those cited in the literature review of this work.

The gaps in research pertaining to the combination of the availability of source code, expansibility of the system and compatibility with the Internet of Things environment are unique to the hardware generated by this system. Due to the specificity of the niche of the kind of hardware generated by this system, the comparison with the full scale personal computer processor is relevant and sufficient to demonstrate that the generator produces synthesizable circuits, and the objective of this work is not to produce the fastest or smallest possible accelerator, but one with acceptable trade-offs and compatible with the Internet of Things.

However, a comparative performance test would be interesting to evaluate the losses or gains in performance in different metrics, and draw inspiration from other design choices, if their code is available at all.

10.2 Conclusions

To properly draw conclusions from this work, we must review the objectives proposed to this research by Chapter 3. There are two different types of objectives set for this work: first, the structural objectives of the entire project, regarding characteristics such as usability of the final generator tool and how it is published; then, there are engineering objectives of the generator system itself, as well those of the generated hardware accelerator. To draw the conclusions of this work, after reiterating each of the main objectives of these two classes, we then state from the main results and system characteristics how these objectives are accomplished, or how future works, in Chapter 11, will be necessary to expand the system in order to do so.

First of all, the resulting tool must be usable by programmers without specific hardware design expertise, and as such this knowledge must be completely isolated by the system and unnecessary to properly use it. We consider that this objective has been reasonably accomplished, but more work can always be done in order to further increase the tool's ease of use. To use the final tool and generate the corresponding VHDL code that when synthesized implements the neural network in hardware, none of the interactions needed from the final user require VHDL programming; all that is needed is to execute the generator system's software, navigate the graphical user interface to select the exported neural network model, and if the system's compatibilities are respected, the code is generated automatically with no further interaction. Furthermore, these restrictions in compatibility are legible by an neural network software programmer, as they are related to machine learning frameworks, topologies, layers and activation functions, which are under the domain of such a professional. However, the limitations in this model of tool are that the generated VHDL code, while synthesizable without modifications, in the case of implementing the accelerator in an FPGA, still require interaction with hardware compilation and synthesis software. Operating this kind of software can be considered specific knowledge, and has some considerations

in regards to configuration, compilation and interpreting the outputs. Still, compiling the code and configuring an FPGA is unavoidable to use this kind of tool, and depending on the specific device and manufacturer, proprietary software is required. Possible potential workarounds to this are discussed in Chapter 11.

Then, considerations involving the continuity and expansion of the project were respected, and thus we consider this objective fulfilled. In the first place, we proposed that the system should be fundamentally modular in order to be easily expandable by the open source development community. Different, independent modules could then be developed, tested and implemented, interacting as little as possible with other separate systems, but still benefitting from a hardware generation common core. Additionally, the modules would be guided by guidelines and compatibility rules, facilitating compatibility and further development. This objective was achieved, as the VHDL code corresponding to neural network topologies, quantization algorithms and neuron activation functions are all modular, guided by design guidelines, and used by a common hardware generation core interpreter. New blocks can be added by following these guidelines and minimally editing the core to include the modifications. In addition, the variables related to the neural network data are concentrated in a single, editable data package file, isolating the other functional blocks from modification by the generator core. This means that each of the functional blocks can be more easily read and understood, and can be simulated and tested individually without modification. Furthermore, the system's expansibility in regards to machine learning frameworks is greatly aided by the inclusion of compatibility with the Open Neural Network Exchange (ONNX) format, which itself is compatible with many other frameworks with several, free and open converters available. As such, with little modifications in the neural network model interpretation, if the framework used can be converted into ONNX, the system is indirectly compatible with it.

Next, we consider that objectives related to the source code's availability and publishing were successful. The system's entire source code, as well as VHDL code respective to all of the system's modules were published in a GitHub repository, under a compatible and popular open source license. Furthermore, the code itself is modular, and the software functional blocks related to the VHDL blocks are isolated and thus more legible. In addition, the GitHub repository includes a `readme` file, which presents

and explains the system, as well as indicating the design guidelines and instructions for its use and expansion by other developers. The chosen open source license, the GNU General Public License v3.0 was chosen for its limitations that protect the developers from warranties and liabilities; as it is permissible enough for private and commercial use, modification and publication; but still offers important conditions over its modification, requiring that all further versions are open and free, with state changes documented and with copy of the same license included. This choice was made to ensure the continuity of the system as an open source system, with care taken to ensure that improvements and further versions continue to be free and open, and limitations to encourage contributions.

To test the neural network model interpreter and automatic hardware accelerator, we need to provide the system with compatible neural networks. Since the system is in its early stages, the activation function and network topology functional block library only includes the Rectified Linear Unit function and the fully-connected neuronal interconnectivity rules. As such, the neural networks to test the system must be designed with these limitations in mind. We consider that the sample networks were successfully designed and trained, and they represented significant and relevant test examples for the system as a whole. The two networks implemented the Exclusive OR logic function and a classifier for the Iris dataset, and were designed and trained until a high accuracy was attained. The Exclusive OR example was considered significant as it outputs linearly inseparable results, and as such, require a multi-layer neural network to be implemented, which was the case. This highlighted the generator system's capability to implement deep learning neural networks. Furthermore, the Iris classifier represented a more realistic example, with a more practical dataset that required a larger neural network model, with more layers and neurons. This test is also significant, as it not only highlights the system's performance when implementing a more complex model, but also we can draw conclusions as to the scalability of the system in regards to its demand of integrated circuit components in the target device, and analyse changes in inference speed and energy expenditure.

Now, concerning engineering requirements for the generated hardware accelerator, we consider that the generated hardware is successful in its proposed objectives. First of all, the generated hardware must fit and thus, be synthesizable in FPGA devices

compatible with the Internet of Things edge computing environment. This was deemed achieved, as the final VHDL code, when compiled by Intel's Quartus Prime software, as shown in Chapter 9, outputted a post-fitting report which indicates that the design fits in the chosen Altera Cyclone IV EP4CE22F17C6N FPGA. The design fits as the compilation and fitting was successful, and the number of demanded integrated circuit components was smaller than their availability in the chosen FPGA. Furthermore, we consider this compatible with the Internet of Things environment as the chosen FPGA is suited for these applications, as it is a small, low-power device specifically made for these implementations. Then, results concerning the power expenditure obtained by the Intel Quartus Prime Power Analyzer tool reports also indicate that the resulting hardware accelerator is compatible with the possible restrictive power availability Internet of Things environment. The order of magnitude of the power expenditure is compatible with lightweight applications using batteries or energy harvesting from small sources, such as miniature solar panels.

Finally, concerning the hardware accelerator performance in regards to its data processing, we also consider this objective to be fulfilled. The VHDL code, when simulated in Modelsim software, shows that its expected logical behavior, when combined with maximum system clock frequency obtained by compilation in Quartus Prime, has significant performance gains when compared with a full-scale high mid range full-scale personal computer Intel i7-8700k CPU. As shown in Chapter 9, the compilation resulted in a system clock frequency, and the logical simulation can be analysed to extract number of clock cycles to certain events. Then, combining clock frequency and number of clock cycles, we can derive time delays until those events. Finally, we conclude the time latency between the first input and its result output by the hardware accelerator, and the time per result once the system stabilized, and we compare these times with those resulting from tests executed in the personal computer CPU. Comparing the results, we conclude that in the tests performed, the FPGA hardware accelerator outperformed the personal computer CPU, showing significant speedup gains, whose importance is compounded by the fact that the full-scale CPU uses around 1000 times more power than the accelerator. This conveys that the generated hardware accelerator is successful in regards to performance throughput speedup and energy expenditure efficiency.

Furthermore, as shown in Chapter 9, the logical simulation results demonstrate

that the outputs obtained by the algorithms implemented in the hardware accelerator closely match those of the model when executed in the original software programming. While the reduction in representation precision due to the quantization of the variables used in the implemented neural networks would pose a reduction in model accuracy, the techniques with which this is done greatly reduce this loss. Firstly, since Tensorflow Lite quantization-aware neural network training is used, the network itself considers the loss in representation accuracy when choosing internal parameters such as weights and biases. During the neural network's training in software, 32 bit floating point variables are used, as training fully in 8 bit integers causes too much accuracy loss and results in ineffective training. However, by including quantization and dequantization steps in between layers, the rounding errors and loss of precision is accurately simulated while preserving the training accuracy. When exported, the model is then fully quantized and can be interpreted by the generator software to execute the arithmetic in 8 bit integers, preserving the expected behavior and the model's accuracy. Consequently, when the network is implemented in hardware, the quantization is already expected and the errors are already taken into account by the model. The quantization algorithm also attempts to maximize representation precision by using zero points and scale parameters, to map the expected variable ranges as precisely as possible into the lower precision, as explained in Chapter 5. Furthermore, the quantization corrections necessary to maintain logical correctness while using these quantized variables in arithmetic operations during the neural network execution were also implemented in their own VHDL functional block, as shown in Chapter 7.

10.2.1 Final conclusions and discussion

In conclusion, an interpreter and generation core system was created that can import and interpret trained artificial neural network models, read their topology, activation functions and trained data such as weights and biases, and by consulting a VHDL functional block library, outputs VHDL code corresponding to a hardware accelerator that implements the inputted model.

The system generated VHDL code that, when compiled and synthesized in an FPGA corresponds to a hardware accelerator using only a trained neural network model as input, without requiring any specialized hardware design knowledge by the

final user.

The resulting accelerator is compatible with restrictions common to Internet of Things edge computing environment, as it can be synthesized using a small, low-power FPGA as target device. In addition, the hardware was tested and compared against a high mid range full-scale personal computer CPU, and showed significant performance gains in regards to execution speed, latency until first result and energy efficiency.

The tests also demonstrated that the hardware's expected behavior matches that of the original neural network model in software, maintaining logical correctness and inference accuracy.

The system is based on a modular function library that follows specific patterns and design guidelines, which facilitates its expansion and maximizes compatibility, and a common interpreter and generator central software core, which is compatible with the Open Neural Network Exchange format, maximizing machine learning development framework interoperability. Furthermore, the core and function library can interpret and generate accelerators for any compatible network imported into the system.

The system then was published in a GitHub repository under the GNU General Public License v3.0, ensuring the source code remains open and free to use, modify and distribute for personal and commercial means, as long as further versions document their state changes and remain open and free under the same license.

11 FURTHER WORK

In this chapter we discuss potential avenues for future work in order to expand the project, whether it be increasing the significance of its tests and results or improving the resulting system itself.

First, while the tests conducted to demonstrate the hardware generation system's functionality were deemed adequate, it would be interesting to include an example which represents a solution that fits the Internet of Things topic, and solves a problem in a realistic and practical way. While the complexity of such a problem could be similar to the Iris classifier used in tests, since it is already a reasonable classification problem with real data and that requires a relatively complex neural network, a real Internet of Things problem would be even more significant.

Another consideration to be made concerns the source code legibility. While code legibility is an abstract problem and difficult to solve and judge objectively, more comments can be added to the system's source code to improve the project's potential longevity and encourage contributions from the open source community.

Also in respect to the project's attractiveness, another potential avenue to improve the tool is to expand its function library. Modern neural network software development uses several different activation functions, neuron interconnectivity rules and preset layers, and including them would make the generator more useful. Then, a more useful generator would further increase interest in using and improving the project, resulting in a virtuous cycle. Furthermore, the system's usability can be better represented by being compatible with pretrained famous neural network models, such as the YOLO classifier (REDMON et al., 2015). As such, an interesting path to improve the system would be to include popular functions used by such systems. Examples are convolutional neural network layers, pooling layers, and hyperbolic tangent activation functions. Furthermore, hardware accelerators of famous neural network models can be included in a pregenerated accelerator library to further facilitate usability and demonstrate the system's functionality.

Another path to increase the resulting system's utility would be to expand its frame-

work compatibility. While the interpreter is compatible with the ONNX format, each framework still needs to be considered individually and converted into ONNX so it can be interpreted. Changes to the interpreter are expected to be minimal, however the converters need to be manually included, and the system could need to be modified to expect small changes in the resulting model caused by the different conversions. Examples of potential framework candidates for inclusion are SciKit learn and Caffe.

One more interesting comment to be made is related to the hardware accelerator scalability. While the tests show that the resulting systems fit in the target hardware with a significant number of components remaining, modern neural network programming often can result in massive networks with many layers and neurons. Since the chosen hardware design architecture chosen was the dedicated circuit design and the resulting hardware's topology mirrors that of the implemented network, bigger networks result in hardware with greater demands in device components. This characterizes a problem in scalability. While in the Internet of Things context, large networks are not expected, it would be interesting that the generator software could generate successful accelerators sacrificing performance instead of generating hardware that is impossible to implement in certain devices, which is the case presently. An interesting avenue to correct this issue would be to draw inspiration from the Deepburning AI project (WANG et al., 2016), that uses "folds" to reuse the same hardware components for different layers or different neurons in the same layer. While this decreases performance lowering inference speed and reducing the effectiveness of the pipelining behavior exploited in our design, by "folding", the neural network operations are queued and thus the behavior of a large network can be implemented in smaller hardware. For example, instead of utilizing a separate generated component for each Rectified Linear Unit layer used in the design, a single component can be generated and the requested operations coming from multiple layers can be queued and executed one after the other, reducing the resulting demand in integrated circuit components. However, to preserve as much performance as possible while still enabling the generation of accelerators for large networks, it would be interesting to offer "folding" as an optional choice, whether it be by asking the final user how many folds are to be made, or by including hardware specifications so the generator can decide by itself to fold as many times as necessary to fit the design in a target hardware.

A comment made while describing the system's usability was that while the generator system outputs VHDL files corresponding to a hardware accelerator, the files still need to be compiled and synthesized for an FPGA using specialized software. This specialized software could be seen as hard to use, and can be considered specialized knowledge, increasing the difficulty to use the system by programmers that do not have FPGA hardware design knowledge. However, this is a hard problem to solve, as different FPGA manufacturers have their own compilation software, which generates files specific for their FPGAs. To solve this issue would be to theoretically program a VHDL compiler compatible with different devices, irrespective of manufacturer, and that would be easier to use than the corresponding companies' compilers. We conclude that this issue would be too complex to solve satisfactorily with little reward, and a potential workaround would be to include files that document simple step-by-step instructions to use the resulting VHDL in an FPGA that could be followed by a programmer with no prior knowledge.

REFERENCES

ABADI, M. et al. **TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems**. 2015. Software available from tensorflow.org. Disponível em: <https://www.tensorflow.org/>. Acesso em: 11/2022.

ABIODUN, O. I. et al. State-of-the-art in artificial neural network applications: A survey. **Heliyon**, v. 4, n. 11, p. e00938, 2018. ISSN 2405-8440. Disponível em: <https://www.sciencedirect.com/science/article/pii/S2405844018332067>. Acesso em: 10/2022.

BAI, J. et al. **ONNX: Open Neural Network Exchange**. [S.l.]: GitHub, 2019. <https://github.com/onnx/onnx>. Acesso em: 11/2022.

BOJARSKI, M. et al. **Explaining How a Deep Neural Network Trained with End-to-End Learning Steers a Car**. 2017.

DAS, D.; SAHOO, L.; DATTA, S. A survey on recommendation system. **International Journal of Computer Applications**, Foundation of Computer Science, v. 160, n. 7, 2017.

DING, Y. et al. A deep learning model to predict a diagnosis of alzheimer disease by using 18f-fdg pet of the brain. **Radiology**, v. 290, n. 2, p. 456–464, 2019. PMID: 30398430. Disponível em: <https://doi.org/10.1148/radiol.2018180958>. Acesso em: 11/2022.

DUA, D.; GRAFF, C. **UCI Machine Learning Repository**. 2017. Disponível em: <http://archive.ics.uci.edu/ml>. Acesso em: 10/2022.

ELNAWAWY, M. et al. Role of fpga in internet of things applications. In: IEEE INTERNATIONAL SYMPOSIUM ON SIGNAL PROCESSING AND INFORMATION TECHNOLOGY (ISSPIT). **Proceeding...** [S.l.: s.n.], 2019. p. 1–6.

ENDSLEY, M. R. Autonomous driving systems: A preliminary naturalistic study of the tesla model s. **Journal of Cognitive Engineering and Decision Making**, v. 11, n. 3, p. 225–238, 2017. Disponível em: <https://doi.org/10.1177/1555343417695197>. Acesso em: 11/2022.

ESMAEILZADEH, H. et al. What is happening to power, performance, and software? **IEEE Micro**, v. 32, n. 3, p. 110–121, 2012.

GITHUB. **GitHub**. 2020. Disponível em: <https://github.com/>. Acesso em: 11/2022.

GORBACHEV, Y. et al. Openvino deep learning workbench: Comprehensive analysis and tuning of neural networks inference. proceeding... In: IEEE/CVF INTERNATIONAL CONFERENCE ON COMPUTER VISION (ICCV) WORKSHOPS. **Procedings...** [S.l.: s.n.], 2019.

HE, K. et al. **Deep Residual Learning for Image Recognition**. 2015.

HORNIK, K.; STINCHCOMBE, M.; WHITE, H. Multilayer feedforward networks are universal approximators. **Neural Netw.**, Elsevier Science, GBR, v. 2, n. 5, p. 359–366, jul 1989. ISSN 0893-6080.

HOWARD, A. G. et al. **MobileNets:** Efficient Convolutional Neural Networks for Mobile Vision Applications. arXiv, 2017. Disponível em: <https://arxiv.org/abs/1704.04861>. Acesso em: 11/2022.

HUBARA, I. et al. Binarized neural networks. In: INTERNATIONAL CONFERENCE ON NEURAL INFORMATION PROCESSING SYSTEMS (NIPS'16), 30. **Proceedings...** NY, USA: Curran Associates, 2016. p. 4114–4122. ISBN 9781510838819.

IANDOLA, F. N. et al. **SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and lt;0.5MB model size**. arXiv, 2016. Disponível em: <https://arxiv.org/abs/1602.07360>. Acesso em: 11/2022.

INTEL. **Intel® Distribution of OpenVINO™ Toolkit**. 2021. Disponível em: <https://software.intel.com/content/www/us/en/develop/tools/openvino-toolkit.html>. Acesso em: 07/2021.

IZEBOUDJEN, N.; LARBES, C.; FARAH, A. A new classification approach for neural networks hardware: from standards chips to embedded systems on chip. **Artificial Intelligence Review**, v. 41, p. 491–534, 2014. Disponível em: <https://doi.org/10.1007/s10462-012-9321-7>. Acesso em: 11/2022.

JACOB, B. et al. **Quantization and Training of Neural Networks for Efficient Integer-Arithmetic-Only Inference**. arXiv, 2017. Disponível em: <https://arxiv.org/abs/1712.05877>. Acesso em: 03/2022.

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

KATHAIL, V. Xilinx vitis unified software platform. In: ACM/SIGDA INTERNATIONAL SYMPOSIUM ON FIELD-PROGRAMMABLE GATE ARRAYS (FPGA'20). **Proceedings...** New York, NY, USA: Association for Computing Machinery, 2020. p. 173–174. ISBN 9781450370998. Disponível em: <https://doi.org/10.1145/3373087.3375887>. Acesso em: 11/2022.

LATTICE sensAI Stack. Lattice Semiconductors. Disponível em: <https://www.latticesemi.com/sensAI>. Acesso em: 11/2022.

LI, F.; ZHANG, B.; LIU, B. **Ternary Weight Networks**. arXiv, 2016. Disponível em: <https://arxiv.org/abs/1605.04711>. Acesso em: 11/2022.

MA, Y. et al. Artificial intelligence applications in the development of autonomous vehicles: a survey. **IEEE/CAA Journal of Automatica Sinica**, v. 7, n. 2, p. 315–329, 2020.

MITCHELL, T. M. **Machine Learning**. New York: McGraw-Hill, 1997. ISBN 978-0-07-042807-2.

MITTAL, S. A survey of fpga-based accelerators for convolutional neural networks. **Neural Computing and Applications**, v. 32, 02 2020.

MITTAL, S.; VETTER, J. S. A survey of methods for analyzing and improving gpu energy efficiency. **ACM Comput. Surv.**, Association for Computing Machinery, New York, NY, USA, v. 47, n. 2, ago. 2014. ISSN 0360-0300. Disponível em: <https://doi.org/10.1145/2636342>. Acesso em: 11/2022.

MOREAU, T. et al. **A Hardware-Software Blueprint for Flexible Deep Learning Specialization**. 2019.

MUYAL, T. **OpenQNPU**. [S.l.]: GitHub, 2022. <https://github.com/ThomasMuyal/OpenQNPU>. Acesso em: 11/2022.

PICCIALLI, F. et al. A survey on deep learning in medicine: Why, how and when? **Information Fusion**, v. 66, p. 111–137, 2021. ISSN 1566-2535. Disponível em: <https://www.sciencedirect.com/science/article/pii/S1566253520303651>. Acesso em: 11/2022.

PLAGWITZ, P. et al. A safari through fpga-based neural network compilation and design automation flows. In: IEEE ANNUAL INTERNATIONAL SYMPOSIUM ON FIELD-PROGRAMMABLE CUSTOM COMPUTING MACHINES (FCCM), 29. **Proceedings...** [S.l.: s.n.], 2021. p. 10–19.

RASTEGARI, M. et al. **XNOR-Net: ImageNet Classification Using Binary Convolutional Neural Networks**. arXiv, 2016. Disponível em: <https://arxiv.org/abs/1603.05279>. Acesso em: 11/2022.

REDMON, J. et al. **You Only Look Once: Unified, Real-Time Object Detection**. arXiv, 2015. Disponível em: <https://arxiv.org/abs/1506.02640>. Acesso em: 11/2022.

REDMON, J.; FARHADI, A. **YOLOv3: An Incremental Improvement**. arXiv, 2018. Disponível em: <https://arxiv.org/abs/1804.02767>. Acesso em: 11/2022.

RUSSAKOVSKY, O. et al. **ImageNet Large Scale Visual Recognition Challenge**. 2015.

RUSSELL, S. J.; NORVIG, P. **Artificial Intelligence: A Modern Approach**. Boston: Pearson, 2021.

WANG, Y.; LI, H.; LI, X. Re-architecting the on-chip memory sub-system of machine-learning accelerator for embedded devices. In: INTERNATIONAL CONFERENCE ON COMPUTER-AIDED DESIGN. NEW YORK, NY, USA: ASSOCIATION FOR COMPUTING MACHINERY (ICCAD '16), 35. **Proceedings...**, 2016. ISBN 9781450344661. Disponível em: <https://doi.org/10.1145/2966986.2967068>. Acesso em: 11/2022.

WANG, Y. et al. Deepburning: Automatic generation of fpga-based learning accelerators for the neural network family. In: ACM/EDAC/IEEE DESIGN AUTOMATION CONFERENCE (DAC), 53. **Proceedings...** [S.l.: s.n.], 2016. p. 1–6.

WU, Y. et al. Google's neural machine translation system: Bridging the gap between human and machine translation. 09 2016.

YOON, Y. H. et al. Intellino: Processor for embedded artificial intelligence. **Electronics**, v. 9, n. 7, 2020. ISSN 2079-9292. Disponível em: <https://www.mdpi.com/2079-9292/9/7/1169>. Acesso em: 11/2022.

APPENDIX A – NUMERICAL DATA FROM SIMULATIONS

Table 13: Iris classifier generated hardware accelerator quantized binary outputs from the Modelsim logical correctness simulation.

Time in picoseconds	Output 0	Output 1	Output 2	Done
1487500	10000011	01111111	10000011	00000001
1907500	10000011	00010111	11011001	00000010
2327500	10000011	01111111	10000011	00000011
2817500	10000011	01111111	10000011	00000100
3097500	10000011	10000011	01110101	00000101
3587500	01111111	10000011	10000011	00000110
3867500	10000011	10000011	01101100	00000111
4357500	01100010	10000101	10000011	00001000
4637500	10000011	01111111	10000011	00001001
5127500	10000011	00010111	11011001	00001010
5547500	01110100	10000011	10000011	00001011
5897500	10000011	01111111	10000011	00001100
6317500	01110000	10000011	10000011	00001101
6737500	01110001	10000011	10000011	00001110
7087500	10000011	10000011	01100111	00001111
7507500	10000011	01111111	10000011	00010000
7997500	10000011	10000011	01101010	00010001
8417500	10000011	01111111	10000011	00010010
8767500	10000011	01111111	10000011	00010011
9187500	01100110	10010010	10000011	00010100
9537500	10000011	01111111	10000011	00010101

Continued on next page

Time in picoseconds	Output 0	Output 1	Output 2	Done
9957500	10000011	10111100	00101110	00010110
10307500	01110010	10000011	10000011	00010111
10727500	10000011	10000011	01110111	00011000
11147500	10000011	10000011	01101100	00011001
11497500	01110010	10000011	10000011	00011010
11917500	01110001	10000011	10000011	00011011
12337500	10000011	10000011	01101011	00011100
12687500	01111010	10000011	10000011	00011101
13107500	10000011	01111111	10000011	00011110
13597500	10000011	10000011	01101011	00011111
13877500	10000011	10000011	01110101	00100000
14367500	10000011	10000011	01101100	00100001
14647500	10000011	10000011	01110001	00100010
15137500	10000011	01111111	10000011	00100011
15417500	10000011	00111011	10110111	00100100
15907500	10000011	10000011	01101100	00100101
16327500	01101110	10000011	10000011	00100110
16747500	10000011	10000011	01101100	00100111
17097500	01110110	10000011	10000011	00101000
17517500	10000011	01111111	10000011	00101001
18007500	01110001	10000011	10000011	00101010
18287500	01110111	10000011	10000011	00101011
18777500	01110110	10000011	10000011	00101100
19197500	01101010	10000011	10000011	00101101
19547500	10000011	10000011	01100111	00101110
19967500	01101111	10000011	10000011	00101111
20317500	01111010	10000011	10000011	00110000
20737500	01101100	10000011	10000011	00110001
21087500	10000011	10000011	01101100	00110010

Table 14: Iris classifier generated hardware accelerator quantized decimal outputs from the Modelsim logical correctness simulation.

Time in picoseconds	Output 0	Output 1	Output 2	Enable
1487500	-125	127	-125	1
1907500	-125	23	-39	2
2327500	-125	127	-125	3
2817500	-125	127	-125	4
3097500	-125	-125	117	5
3587500	127	-125	-125	6
3867500	-125	-125	108	7
4357500	98	-123	-125	8
4637500	-125	127	-125	9
5127500	-125	23	-39	10
5547500	116	-125	-125	11
5897500	-125	127	-125	12
6317500	112	-125	-125	13
6737500	113	-125	-125	14
7087500	-125	-125	103	15
7507500	-125	127	-125	16
7997500	-125	-125	106	17
8417500	-125	127	-125	18
8767500	-125	127	-125	19
9187500	102	-110	-125	20
9537500	-125	127	-125	21
9957500	-125	-68	46	22
10307500	114	-125	-125	23
10727500	-125	-125	119	24
11147500	-125	-125	108	25
11497500	114	-125	-125	26
11917500	113	-125	-125	27
12337500	-125	-125	107	28
12687500	122	-125	-125	29
13107500	-125	127	-125	30

Continued on next page

Time in picoseconds	Output 0	Output 1	Output 2	Enable
13597500	-125	-125	107	31
13877500	-125	-125	117	32
14367500	-125	-125	108	33
14647500	-125	-125	113	34
15137500	-125	127	-125	35
15417500	-125	59	-73	36
15907500	-125	-125	108	37
16327500	110	-125	-125	38
16747500	-125	-125	108	39
17097500	118	-125	-125	40
17517500	-125	127	-125	41
18007500	113	-125	-125	42
18287500	119	-125	-125	43
18777500	118	-125	-125	44
19197500	106	-125	-125	45
19547500	-125	-125	103	46
19967500	111	-125	-125	47
20317500	122	-125	-125	48
20737500	108	-125	-125	49
21087500	-125	-125	108	50

Table 15: Iris classifier generated neural network expected outputs.

Output 0	Output 1	Output 2
-125	127	-125
-125	23	-39
-125	127	-125
-125	127	-125
-125	-125	117
127	-125	-125
-125	-125	108
116	-125	-125
-125	127	-125
-125	23	-39
115	-125	-125
-125	127	-125
112	-125	-125
113	-125	-125
-125	-125	103
-125	127	-125
-125	-125	106
-125	127	-125
-125	127	-125
109	-122	-125
-125	127	-125
-125	-68	46
108	-125	-125
-125	-125	119
-125	-125	113
125	-125	-125
112	-125	-125
-125	-125	107
122	-125	-125
-125	127	-125
-125	-125	107

Continued on next page

Output 0	Output 1	Output 2
-125	-125	117
-125	-125	113
-125	-125	113
-125	127	-125
-125	68	-83
-125	-125	108
110	-125	-125
-125	-125	108
122	-125	-125
-125	127	-125
113	-125	-125
122	-125	-125
120	-125	-125
106	-125	-125
-125	-125	103
112	-125	-125
118	-125	-125
109	-125	-125
-125	-125	108

Table 16: Iris test dataset inputs.

Input 0	Input 1	Input 2	Input 3
6	3.4	4.5	1.6
5.9	3.2	4.8	1.8
6.3	2.3	4.4	1.3
6.2	2.9	4.3	1.3
6.2	3.4	5.4	2.3
4.4	3.2	1.3	0.2
6.9	3.2	5.7	2.3
5	3	1.6	0.2
5	2	3.5	1
6	3	4.8	1.8
5.8	4	1.2	0.2
5.5	2.6	4.4	1.2
5.2	3.5	1.5	0.2
5.7	4.4	1.5	0.4
5.9	3	5.1	1.8
5.6	3	4.5	1.5
7.7	2.6	6.9	2.3
6.6	3	4.4	1.4
5.6	2.5	3.9	1.1
5	3.5	1.6	0.6
7	3.2	4.7	1.4
6.3	2.8	5.1	1.5
4.4	2.9	1.4	0.2
6.3	2.5	5	1.9
6.7	2.5	5.8	1.8
4.4	3	1.3	0.2
5	3.4	1.6	0.4
7.7	3.8	6.7	2.2
4.9	3	1.4	0.2
5.9	3	4.2	1.5
7.7	3	6.1	2.3
6.8	3	5.5	2.1

Continued on next page

Input 0	Input 1	Input 2	Input 3
6.4	2.8	5.6	2.2
6.4	2.8	5.6	2.1
6.3	3.3	4.7	1.6
6.2	2.2	4.5	1.5
7.2	3.2	6	1.8
5.1	3.7	1.5	0.4
6.7	3.3	5.7	2.5
4.9	3.1	1.5	0.1
5.8	2.7	3.9	1.2
5.5	3.5	1.3	0.2
4.7	3.2	1.3	0.2
4.6	3.4	1.4	0.3
4.8	3	1.4	0.3
5.6	2.8	4.9	2
5.4	3.9	1.3	0.4
4.8	3	1.4	0.1
5.4	3.9	1.7	0.4
7.4	2.8	6.1	1.9

Source: Dua e Graff (2017)

Table 17: Iris classifier generated hardware accelerator quantized decimal inputs for the Modelsim logical correctness simulation.

Input 0	Input 1	Input 2	Input 3	Enable
70	-15	21	-75	1
67	-22	30	-68	2
80	-51	17	-84	3
77	-31	14	-84	4
77	-15	50	-51	5
17	-22	-84	-121	6
100	-22	60	-51	7
37	-28	-75	-121	8
37	-61	-12	-94	9
70	-28	30	-68	10
64	4	-88	-121	11
54	-41	17	-88	12
44	-12	-78	-121	13
60	17	-78	-114	14
67	-28	40	-68	15
57	-28	21	-78	16
127	-41	100	-51	17
90	-28	17	-81	18
57	-45	1	-91	19
37	-12	-75	-108	20
103	-22	27	-81	21
80	-35	40	-78	22
17	-31	-81	-121	23
80	-45	37	-65	24
93	-45	64	-68	25
17	-28	-84	-121	26
37	-15	-75	-114	27
127	-2	93	-55	28
34	-28	-81	-121	29
67	-28	11	-78	30

Continued on next page

Input 0	Input 1	Input 2	Input 3	Enable
127	-28	74	-51	31
97	-28	54	-58	32
83	-35	57	-55	33
83	-35	57	-58	34
80	-18	27	-75	35
77	-55	21	-78	36
110	-22	70	-68	37
40	-5	-78	-114	38
93	-18	60	-45	39
34	-25	-78	-124	40
64	-38	1	-88	41
54	-12	-84	-121	42
27	-22	-84	-121	43
24	-15	-81	-118	44
30	-28	-81	-118	45
57	-35	34	-61	46
50	1	-84	-114	47
30	-28	-81	-124	48
50	1	-71	-114	49
117	-35	74	-65	50

Table 18: Iris classifier generated hardware accelerator quantized binary inputs for the Modelsim logical correctness simulation.

Time in picoseconds	Input 0	Input 1	Input 2	Input 3	Enable
800000	01000110	11110001	00010101	10110101	00000001
1200000	01000011	11101010	00011110	10111100	00000010
1600000	01010000	11001101	00010001	10101100	00000011
2000000	01001101	11100001	00001110	10101100	00000100
2400000	01001101	11110001	00110010	11001101	00000101
2800000	00010001	11101010	10101100	10000111	00000110
3200000	01100100	11101010	00111100	11001101	00000111
3600000	00100101	11100100	10110101	10000111	00001000
4000000	00100101	11000011	11110100	10100010	00001001
4400000	01000110	11100100	00011110	10111100	00001010
4800000	01000000	00000100	10101000	10000111	00001011
5200000	00110110	11010111	00010001	10101000	00001100
5600000	00101100	11110100	10110010	10000111	00001101
6000000	00111100	00010001	10110010	10001110	00001110
6400000	01000011	11100100	00101000	10111100	00001111
6800000	00111001	11100100	00010101	10110010	00010000
7200000	01111111	11010111	01100100	11001101	00010001
7600000	01011010	11100100	00010001	10101111	00010010
8000000	00111001	11010011	00000001	10100101	00010011
8400000	00100101	11110100	10110101	10010100	00010100
8800000	01100111	11101010	00011011	10101111	00010101
9200000	01010000	11011101	00101000	10110010	00010110
9600000	00010001	11100001	10101111	10000111	00010111
10000000	01010000	11010011	00100101	10111111	00011000
10400000	01011101	11010011	01000000	10111100	00011001
10800000	00010001	11100100	10101100	10000111	00011010
11200000	00100101	11110001	10110101	10001110	00011011
11600000	01111111	11111110	01011101	11001001	00011100
12000000	00100010	11100100	10101111	10000111	00011101
12400000	01000011	11100100	00001011	10110010	00011110

Continued on next page

Time in picoseconds	Input 0	Input 1	Input 2	Input 3	Enable
12800000	01111111	11100100	01001010	11001101	00011111
13200000	01100001	11100100	00110110	11000110	00100000
13600000	01010011	11011101	00111001	11001001	00100001
14000000	01010011	11011101	00111001	11000110	00100010
14400000	01010000	11101110	00011011	10110101	00100011
14800000	01001101	11001001	00010101	10110010	00100100
15200000	01101110	11101010	01000110	10111100	00100101
15600000	00101000	11111011	10110010	10001110	00100110
16000000	01011101	11101110	00111100	11010011	00100111
16400000	00100010	11100111	10110010	10000100	00101000
16800000	01000000	11011010	00000001	10101000	00101001
17200000	00110110	11110100	10101100	10000111	00101010
17600000	00011011	11101010	10101100	10000111	00101011
18000000	00011000	11110001	10101111	10001010	00101100
18400000	00011110	11100100	10101111	10001010	00101101
18800000	00111001	11011101	00100010	11000011	00101110
19200000	00110010	00000001	10101100	10001110	00101111
19600000	00011110	11100100	10101111	10000100	00110000
20000000	00110010	00000001	10111001	10001110	00110001
20400000	01110101	11011101	01001010	10111111	00110010