



UNIVERSITY OF THESSALY
SCHOOL OF ENGINEERING DEPARTMENT OF
ELECTRICAL AND COMPUTER ENGINEERING

Implementation of Low Cost Neural Networks

Dimploma Thesis

Fotios Tsipis

Supervisor: Karakonstantis Georgios

Volos, September 2025



UNIVERSITY OF THESSALY
SCHOOL OF ENGINEERING DEPARTMENT OF
ELECTRICAL AND COMPUTER ENGINEERING

Implementation of Low Cost Neural Networks

Diploma Thesis

Fotios Tsipis

Supervisor: Karakonstantis Georgios

Volos, September 2025



ΠΑΝΕΠΙΣΤΗΜΙΟ ΘΕΣΣΑΛΙΑΣ

**ΠΟΛΥΤΕΧΝΙΚΗ ΣΧΟΛΗ ΤΜΗΜΑ ΗΛΕΚΤΡΟΛΟΓΩΝ
ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΟΣ ΥΠΟΛΟΓΙΣΤΩΝ**

Υλοποίηση Νευρωνικών Δικτύων Χαμηλού Κόστους

Διπλωματική Εργασία

Φώτιος Τσίπης

Επιβλέπων: Καρακωνσταντής Γεώργιος

Βόλος, Σεπτέμβριος 2025

Εγκρίνεται από την Επιτροπή Εξέτασης:

Επιβλέπων: **Καρακωνσταντής Γεώργιος**

Αναπληρωτής Καθηγητής, Τμήμα Ηλεκτρολόγων Μηχανικών
και Μηχανικών Υπολογιστών, Πανεπιστήμιο Θεσσαλίας

Μέλος: **Σταμούλης Γεώργιος**

Καθηγητής, Τμήμα Ηλεκτρολόγων Μηχανικών και Μηχανικών
Υπολογιστών, Πανεπιστήμιο Θεσσαλίας

Μέλος: **Ευμορφόπουλος Νέστωρ**

Καθηγητής, Τμήμα Ηλεκτρολόγων Μηχανικών και Μηχανικών
Υπολογιστών, Πανεπιστήμιο Θεσσαλίας

DISCLAIMER ON ACADEMIC ETHICS AND INTELLECTUAL PROPERTY RIGHTS

Fully conscious of copyright regulations, I hereby declare that this diploma thesis, together with any electronic files and source codes created or adapted during its preparation, is exclusively the result of my own effort. It does not infringe upon the intellectual property rights, personal rights, or personal data of third parties, nor does it include material from others for which authorization would be required. The work is not a product of total or partial plagiarism, and the external sources that have been used are limited strictly to the bibliographic references, cited according to academic standards. Wherever ideas, text, files, or material from other authors have been incorporated, these are explicitly acknowledged in the text and listed in full in the bibliography. I further affirm that the outcomes of this thesis have not been submitted in pursuit of any other academic degree. I accept full personal responsibility for any legal or administrative consequences that may arise if, at any time, this thesis or part of it is shown not to be my own original work due to plagiarism.

The Declarant

Fotios Tsipis

ΕΥΧΑΡΙΣΤΙΕΣ

Θα ήθελα να εκφράσω τις θερμές μου ευχαριστίες στον επιβλέποντα καθηγητή μου, κ. Γεώργιο Καρακωνσταντή, για την καθοδήγηση, την πολύτιμη βοήθεια και την άμεση ανταπόκρισή του κατά τη διάρκεια της διπλωματικής μου εργασίας, καθώς επίσης και στους Καθηγητές κ. Σταμούλη Γεώργιο και κ. Ευμορφόπουλο Νέστορ για τη συμβολή τους αξιολόγηση της παρούσας εργασίας. Ευχαριστώ θερμά την οικογένειά μου για τη συνεχή στήριξη και ενθάρρυνση καθ' όλη τη διάρκεια των σπουδών μου, ενώ νιώθω ιδιαίτερη ευγνωμοσύνη για όλους όσους συνέβαλαν, άμεσα ή έμμεσα, στην ολοκλήρωση αυτής της προσπάθειας, προσφέροντάς μου γνώσεις, υποστήριξη και έμπνευση.

Implementation of Low Cost Neural Networks

Fotios Tsipis

Abstract

The thesis focuses on the design of Neural Networks in hardware for the detection of cardiac arrhythmias from electrocardiogram (ECG) data. Arrhythmias are serious cardiac conditions that require early diagnosis and intervention. The main objective of the thesis is to reduce the cost of the Neural Network to allow its implementation in resource-constrained systems, such as handheld devices or medical diagnostic tools. The implementation of this technology in low-cost systems will greatly facilitate the detection of cardiac arrhythmias as it can be performed by battery-powered systems.

To achieve this goal, several Neural Network models will be tested and trained from the MIT-BIH Arrhythmia Database, which includes electrocardiographic signals from a large number of patients and is widely used in automatic cardiac arrhythmia analysis applications. The model training is performed in Python code, using the TensorFlow and Keras libraries, while its conversion to hardware description language (HDL) is done using the HLS4ML tool, which allows Verilog code generation from pre-trained Neural Networks. The Verilog code is then imported and evaluated in the Vivado software, with the aim of analyzing parameters such as space consumption, timing and power, as they have an important role in the implementation of resource-constrained systems.

To reduce the cost of Neural Networks, the quantization technique is considered at various levels of accuracy (12, 8, 4 and 2 bit), as well as the combination of pruning and quantization in implementable models. This techniques allows reducing the size of the Neural Networks without significantly sacrificing the model performance. The thesis further analyzes the impact of quantization and pruning on model accuracy as well as on area, timing, and power consumption of the Neural Network. Through this analysis, the paper aims to optimize Neural Networks for use in resource-constrained systems, such as portable cardiac data monitors or embedded systems for real-time medical diagnostics.

Keywords: Artificial Intelligence, Convolutional Neural Network, Low-Cost Hardware Implementation, MIT-BIH Arrhythmia Database, HLS4ML, Python, Vivado, Quantization, Area Analysis, Timing Analysis, Power Analysis

Υλοποίηση Νευρωνικών Δικτύων Χαμηλού Κόστους

Φώτιος Τσίπης

Περίληψη

Η διπλωματική εστιάζει στην σχεδίαση Νευρωνικού Δικτύου σε υλικό για την ανίχνευση καρδιακών αρρυθμιών από δεδομένα ηλεκτροκαρδιογραφήματος (ECG). Οι αρρυθμίες αποτελούν σοβαρές καρδιακές παθήσεις που απαιτούν έγκαιρη διάγνωση και παρέμβαση. Κύριος στόχος της διπλωματικής είναι η μείωση του κόστους του Νευρωνικού Δικτύου, ώστε να επιτραπεί η υλοποίηση του σε συστήματα με περιορισμένους πόρους, όπως φορητές συσκευές ή ιατρικά διαγνωστικά εργαλεία. Η εφαρμογή αυτής της τεχνολογίας σε συστήματα χαμηλού κόστους θα διευκολύνει σημαντικά την ανίχνευση καρδιακών αρρυθμιών, καθώς θα μπορεί να γίνεται από συστήματα που λειτουργούν με μπαταρίες.

Για την επίτευξη αυτού του στόχου θα δοκιμαστούν διάφορα μοντέλα Νευρωνικών Δικτύων τα οποία θα εκπαιδεύονται από την βάση δεδομένων MIT-BIH Arrhythmia Database, η οποία περιλαμβάνει ηλεκτροκαρδιογραφικά σήματα από πλήθος ασθενών και χρησιμοποιείται ευρέως σε εφαρμογές αυτόματης ανάλυσης καρδιακών αρρυθμιών. Η εκπαίδευση του μοντέλου πραγματοποιείται σε κώδικα Python, χρησιμοποιώντας τις βιβλιοθήκες TensorFlow και Keras, ενώ η μετατροπή του σε γλώσσα περιγραφής υλικού (HDL) γίνεται μέσω του εργαλείου HLS4ML, το οποίο επιτρέπει την παραγωγή κώδικα Verilog από προεκπαιδευμένα Νευρωνικά Δίκτυα. Στην συνέχεια ο κώδικας Verilog εισάγεται και αξιολογείται στο λογισμικό Vivado, με στόχο την ανάλυση παραμέτρων όπως η κατανάλωση χώρου, η καθυστέρηση και η ισχύς, καθώς έχουν σημαντικό ρόλο στην υλοποίηση συστημάτων με περιορισμένους πόρους.

Για την μείωση του κόστους των Νευρωνικών Δικτύων εξετάζεται η τεχνική quantization σε διάφορα επίπεδα ακρίβειας (12, 8, 4 και 2 bits), καθώς και ο συνδιασμός pruning και quantization σε υλοποιήσιμα μοντέλα. Οι τεχνικές αυτές επιτρέπουν τη μείωση του μεγέθους του Νευρωνικού Δικτύου, χωρίς να θυσιάζεται σημαντικά η απόδοση του μοντέλου. Η διπλωματική αναλύει περαιτέρω τον αντίκτυπο του quantization και του pruning στην ακρίβεια του μοντέλου καθώς και στις παραμέτρους χώρου, χρονισμού και κατανάλωσης ισχύος του Νευρωνικού Δικτύου. Μέσω αυτής της ανάλυσης, η εργασία στοχεύει στη βελτιστοποίηση των Νευρωνικών Δικτύων για χρήση σε συστήματα με περιορισμένους πόρους, όπως οι φορητές συσκευές παρακολούθησης καρδιολογικών δεδομένων ή τα ενσωματωμένα συστήματα για ιατρικές διαγνώσεις σε πραγματικό χρόνο.

Λέξεις-κλειδιά: Τεχνητή Νοημοσύνη, Συνελκτικό Νευρωνικό Δίκτυο, Χαμηλού Κόστους Υλοποίηση Υλικού, MIT-BIH Arrhythmia Database, HLS4ML, Python, Vivado, Quantization, Ανάλυση Χώρου, Ανάλυση Χρόνου, Ανάλυση Ισχύς.

Table of Contents

Abstract.....	vi
Περίληψη.....	vii
Table of Contents.....	viii
List of Figures.....	x
List of Tables.....	xii
Chapter 1: Introduction.....	1
Chapter 2: AI and Neural Networks: Foundations, Cardiac Health Applications, and Model Optimization.....	3
2.1 Background on AI/ML/NNs.....	3
2.1.1 Artificial Intelligence (AI).....	3
2.1.2 Machine Learning (ML).....	4
2.1.3 Neural Networks (NNs).....	8
2.2 Arrhythmia Detection with Neural Networks.....	14
2.2.1 Recent Technological Approaches and Notable Studies.....	15
2.2.1.1 Convolutional Neural Networks for ECG Analysis.....	15
2.2.1.2 Long Short-Term Memory Networks for ECG Analysis.....	17
2.2.1.3 Hybrid and Advanced Neural Network Architectures.....	18
2.2.2 Resource Barriers to Real-World ECG-AI.....	20
2.3 Neural Networks Size Reduction Techniques.....	21
2.3.1 Quantization.....	21
2.3.2 Pruning.....	24
2.3.2.1 Unstructured Pruning.....	24
2.3.2.2 Structured Pruning.....	25
2.3.3 Low-Rank Factorization.....	25
2.3.4 Knowledge Distillation.....	26
Chapter 3: Neural Network Hardware Implementation Flow.....	28
3.1 Capabilities.....	29
3.2 Core Principles.....	30
3.3 Internal.....	31
3.4 Python API Configuration.....	34
Chapter 4: Implementation of Cardiac Arrhythmia Detection Models.....	36
4.1 Codebase Selection.....	36
4.1.1 Codebase Analysis.....	36
4.1.2 Core Modifications.....	37
4.2 CNN Models and Frameworks.....	39
4.3 HLS4ML Implementation.....	45
4.3.1 Configuration and Model Conversion.....	45
4.3.2 Quantization Implementation.....	48
4.4 Vivado.....	54
4.5 Timing Improvement.....	60
Chapter 5: Results.....	63
5.1 Hardware Implementation Results.....	63
5.1.1 Quantization 1.....	64
5.1.2 Quantization 2.....	65
5.1.3 Quantization 3.....	67
5.2 Performance Analysis and Design Trade-offs.....	68

5.2.1 Accuracy.....	68
5.2.2 Timing.....	71
5.2.3 Utilization.....	73
5.2.4 Power.....	77
5.2.5 Timing Improvement.....	83
5.2.5.1 Timing Results from Clock Optimization.....	85
5.2.5.2 Utilization Results from Clock Optimization.....	86
5.2.5.3 Power Results from Clock Optimization.....	87
Chapter 6: Conclusions.....	91
References.....	92

List of Figures

Figure 1: Machine Learning in relation to Artificial Intelligence ^[9]	5
Figure 2: Automatic generation of text captions for images with Deep Networks. ^[10] ...	6
Figure 3: Example of topic modeling, a method used for document analysis. ^[10]	7
Figure 4: Non-linear model of an Artificial Neuron. ^[11]	10
Figure 5: (a) Threshold Function (b) Sigmoid function for varying ^[11]	11
Figure 6: Feedforward Network with a Single Layer of neurons ^[11]	13
Figure 7: Fully connected Multilayer Feedforward Network with one Hidden Layer. ^[11]	14
Figure 8: The parameters used in configuring the LDCNN model for the MIT-BIH Arrhythmia dataset. ^[15]	16
Figure 9: The parameters used in configuring the LDCNN model for the PTB diagnostic ECG dataset. ^[15]	17
Figure 10: Performance metrics of deep learning models. ^[16]	18
Figure 11: Comparison of LSTM DL model with previous models using cardiac arrhythmia dataset. ^[16]	18
Figure 12: The performance of the proposed method compared to previous methods on the MIT-BIH dataset. ^[19]	19
Figure 13: The process of model quantization ^[22]	23
Figure 14: A taxonomy of quantization based on the focus of each method. ^[22]	24
Figure 15: The structure of a neural network after pruning. ^[22]	25
Figure 16: The core concept of low-rank decomposition. ^[22]	26
Figure 17: The process of model distillation. ^[22]	27
Figure 18: Overview of the HLS4ML workflow, converting trained machine learning models into optimized hardware designs for FPGA deployment. ^[30]	29
Figure 19: Model 1.....	39
Figure 20: Model 1 accuracy after evaluation.....	40
Figure 21: Model 2.....	41
Figure 22: Model 2 accuracy after evaluation.....	42
Figure 23: Model 3.....	42
Figure 24: Model 3 accuracy after evaluation.....	42
Figure 25: Model 4.....	44
Figure 26: Model 4 accuracy after training and pruning.....	44
Figure 27: Diagram of Model 1 after conversion to HLS model.....	46
Figure 28: Create Block Design.....	54
Figure 29: Import IP Block.....	55
Figure 30: Add the IP Block to the Diagram.....	55
Figure 31: Modules connections.....	56
Figure 32: Clock Settings.....	56
Figure 33: Reset Settings.....	56
Figure 34: Output Data Settings.....	57
Figure 35: Result validation Settings.....	57
Figure 36: Final Block Design.....	58
Figure 37: Run Synthesis and Run Implementation buttons.....	59
Figure 38: Report Timing Summary, Report Power and Report Utilization buttons...	60
Figure 39: Clocking Wizard IP Module.....	60
Figure 40: Output Clocks tab.....	61

Figure 41: Requested Output Freq and OK button.....	62
Figure 42: Model 1 accuracy.....	69
Figure 43: Model 2 accuracy.....	69
Figure 44: Model 3 accuracy.....	70
Figure 45: Model 4 accuracy.....	71
Figure 46: Timing trade offs for Quantization 1.....	72
Figure 47: Time trade offs for Quantization 2.....	72
Figure 48: Time trade offs for Quantization 3.....	73
Figure 49: Utilization trade offs for Quantization 1.....	75
Figure 50: Utilization trade offs for Quantization 2.....	75
Figure 51: Utilization trade offs for Quantization 3.....	76
Figure 52: LUTs-Accuracy Trade-offs for Model 3.....	77
Figure 53: LUTs-Accuracy Trade-offs for Model 4.....	77
Figure 54: Total, Dynamic and Static Power trade offs for Quantization 1.....	79
Figure 55: Clock, Signal, Logic, BRAM, URAM and DSP Power trade offs for Quantization 1.....	79
Figure 56: Total, Dynamic and Static Power trade offs for Quantization 2.....	80
Figure 57: Clock, Signal, Logic, BRAM, URAM and DSP Power trade offs for Quantization 2.....	80
Figure 58: Total, Dynamic and Static Power trade offs for Quantization 3.....	81
Figure 59: Clock, Signal, Logic, BRAM, URAM and DSP Power trade offs for Quantization 3.....	81
Figure 60: Total Power-Accuracy Trade-offs for Model 3.....	82
Figure 61: Total Power-Accuracy Trade-offs for Model 4.....	83
Figure 62: WNS and WPWS trade offs from Clock Optimization.....	85
Figure 63: WHS trade offs for Clock Optimization.....	86
Figure 64: Total, Dynamic and Static Power trade offs from the Clock Optimization.	88
Figure 65: Clock, Signal, Logic, BRAM, URAM and DSP Power trade off of the Clock Optimization.....	89
Figure 66: Total Power and Delay Trade-offs for Model 3 Optimized and Model 4 Optimized.....	90

List of Tables

Table 1: Timing Report for Quantization 1 in nano-seconds.....	65
Table 2: Power Report for Quantization 1 in Watts.....	65
Table 3: Utilization Report for Quantization 1.....	65
Table 4: Timing Report for Quantization 2 in nano-seconds.....	66
Table 5: Power Report in Quantization on Watts.....	66
Table 6: Utilization Report for Quantization 2.....	67
Table 7: Timing Report for Quantization 3 in nano-seconds.....	67
Table 8: Power Report for Quantization 3 in Watts.....	68
Table 9: Utilization Report for Quantization 3.....	68
Table 10: Timing Results after Clocking Optimization.....	84
Table 11: Utilization Results after Clocking Optimization.....	84
Table 12: Power Results after Clocking Optimization.....	85
Table 13: LUTs trade offs from Clock Optimization.....	87
Table 14: Total Power and the Percent Raise of Clock Optimized Models.....	87
Table 15: Results of the Power Report after the Clock Optimization.....	88

Chapter 1: Introduction

Cardiovascular diseases are one of the leading cause of mortality worldwide, with cardiac arrhythmias constituting a significant subset of these conditions that can lead to serious complications if not detected and managed early.^[40] The early diagnosis of arrhythmias plays a crucial role in preventing adverse cardiac events and improving patient outcomes. Recent advances in Artificial Intelligence (AI) have demonstrated remarkable potential in automating the detection and prediction of cardiovascular diseases through analysis of biomedical signals such as electrocardiograms (ECG) ^[1]. Neural Networks (NNs), due to their ability to model complex, non-linear relationships within data, have been extensively applied for arrhythmia classification with promising accuracy.

However, despite the success of these AI models, their practical application in real-world clinical or portable devices is hindered by the substantial computational resources required. State-of-the-art Neural Networks are often deep and complex, resulting in high demands on memory, processing power, and energy consumption, which pose significant challenges for deployment in embedded systems or handheld medical devices ^{[2][3]}. This limitation is particularly critical in scenarios requiring real-time monitoring and analysis, where latency and power efficiency are essential.

To bridge this gap, numerous efforts have focused on optimizing Neural Network models to reduce their computational and hardware footprints, enabling efficient deployment on resource-constrained platforms such as Field Programmable Gate Arrays (FPGAs) or Application-Specific Integrated Circuits (ASICs). Optimization strategies include pruning, compression, and quantization, among which quantization (the reduction of numerical precision of network parameters and activations) has gained considerable attention due to its ability to drastically reduce model size and power consumption while maintaining acceptable accuracy ^{[4][5]}. These techniques facilitate the creation of lightweight models suitable for embedded medical applications without sacrificing diagnostic performance.

This thesis addresses the design and hardware implementation of Neural Networks for cardiac arrhythmia detection, with a primary focus on reducing implementation cost to support operation within resource-limited environments. The MIT-BIH Arrhythmia Database is used for the model training. Then, the research employs the HLS4ML tool to turn trained Neural Networks into synthesizable Verilog code. The study investigates the effects of applying quantization at varying bits (12, 8, 4 and 2 bits), also combined with the pruning technique, on model accuracy and how it affected hardware metrics including area, timing, and power consumption using Vivado's software to evaluate them.

Specifically, Chapter 2 reviews the theoretical foundations of AI/ML/NNs, surveys NNs approaches to arrhythmia detection, outlines practical resource barriers, and introduces the key size-reduction methods of quantization, pruning, knowledge distillation and low-rank factorization. Chapter 3 presents HLS4ML's purpose, features, configuration concepts and Python API. Chapter 4 details the the implementation pipeline: selection and adaptation of an open-source arrhythmia codebase, design of four CNN variants, model conversion and quantization workflows in HLS4ML, and synthesis/timing optimization in Vivado. Chapter 5

reports the experimental results and design trade-offs across quantization and pruning, analyzing accuracy, timing/latency, resource utilization, and power, including effects of clock-optimization.

Chapter 2: AI and Neural Networks: Foundations, Cardiac Health Applications, and Model Optimization

2.1 Background on AI/ML/NNs

This chapter provides the necessary theoretical background for understanding the technologies used in this Diploma Thesis. Initially, the fundamental concepts of Artificial Intelligence, Machine Learning, and Neural Networks are presented. Subsequently, the focus shifts to the application of Neural Networks for the detection of cardiac diseases. Finally, the techniques applied to reduce the cost and size of these models are analyzed, so that their implementation in resource-constrained systems is feasible.

2.1.1 Artificial Intelligence (AI)

The term Artificial Intelligence (AI) refers to the field of interdisciplinary research that examines the ways in which it is possible to create machines (i.e., computers comprising both hardware and software) that attempt to solve complex problems in a manner that can be characterized as intelligence. Due to the abstract nature of intelligence, it is not surprising that the definition of what constitutes AI is often unclear and evolves over time. One of the sub-fields of AI is Machine Learning known as ML. The term ML refers to algorithms that improve through experience, rather than operating based on predefined rules as was the case before the development of the AI field. A particularly significant and growing category of ML is Deep Learning (DL), which focuses on algorithms that use multi-layered Neural Networks (NN). The inspiration for these algorithms stems from the human brain and the functioning of its nervous system.^[6]

The idea of machines exhibiting intelligence was decisively influenced by developments in neurology in the 1940s, which described the brain as an electrical network of neurons, as well as by the pioneering work of Alan Turing on computational processes. In 1950, Turing posed the question, "Can machines think?" and proposed the well-known "Turing Test" as a practical experiment to determine if a machine could imitate human behavior. Artificial Intelligence was officially established as an academic field in the summer of 1956, at a conference at Dartmouth College organized by John McCarthy. There, the "Logic Theorist" program by Alan Newell and Herbert Simon was presented, and the participants agreed on making use of the title "Artificial Intelligence" to describe the new field.^[6]

Despite the initial optimism, early artificial intelligence programs encountered serious difficulties with problems that required more than simple trial-and-error strategies. Ambitious projects, such as automatic translation, failed, leading the governments of the USA and the United Kingdom to withdraw significant funding. This period of reduced expectations and diminished funding, from 1974 to 1980, was named the first "AI winter." Artificial Intelligence experienced a resurgence in the 1980s with "expert systems," which solved specialized problems based on rules derived from human experience. However, their inability to learn, combined with

high maintenance costs, led to a decline in interest and to a second "winter" in the late 1980s^[6].

A resurgence occurred in the 1990s, with the increase in computational power and the focus on specialized applications. A milestone was the victory of IBM's Deep Blue computer over the world chess champion Garry Kasparov in 1997. This achievement, although impressive, also gave rise to doubts, reflecting the "AI effect" phenomenon: tasks that are initially considered indicators of intelligence lose their significance once they are achieved by computers^[6].

The modern era of Artificial Intelligence, since the early 2000s, is characterized by three key factors that have accelerated its evolution. First, the explosive growth in the volume of available data (big data) has fueled the progress of Machine Learning, as the performance of models directly depends on the quantity and quality of the training data. Second, the exponential increase in computational power, mainly through the use of graphics processing units (GPUs), has made the faster implementation of complex algorithms feasible compared to traditional processors (CPUs). Third, the development of open-source libraries such as TensorFlow and PyTorch, as well as affordable cloud platforms, has democratized access to advanced AI tools. As a result, Artificial Intelligence has been integrated into countless aspects of daily life, from search engines and spam detection systems to facial recognition applications, autonomous vehicles, and the recognition of health problems.^[6]

One more reason that the AI has gain so much popularity is that AI models can be deployed on specialized hardware accelerators such as Field Programmable Gate Arrays (FPGA). FPGAs are hardware accelerators that are increasingly gaining popularity in the implementation of custom models for AI, particularly in applications with limited resources. Unlike GPUs, which are fixed in functionality and optimized for general parallel computing, FPGAs provide the unique ability to be reprogrammed for specific tasks, offering significant flexibility in how neural networks are implemented.^[47] In particular, FPGAs offer superior energy efficiency compared to GPUs for certain workloads, making them an attractive choice for embedded systems and mobile devices that require optimized power consumption without compromising performance.^{[47][48]}

2.1.2 Machine Learning (ML)

Machine Learning (ML) is a central subfield of Artificial Intelligence that deals with the development of algorithms that can learn from data and make predictions about it. One of the pioneers of the field, Arthur Samuel, defined Machine Learning in 1959 as "the field of study that gives computers the ability to learn without being explicitly programmed."^[7] This definition remains the most widespread to this day. Later, in 1997, Tom M. Mitchell offered a more formalized definition, which is widely used in the academic community: "Machine learning is present when a system, after being exposed to experience E, achieves better results in a group of tasks T, as evaluated by a chosen performance criterion P." This approach gave a more measurable dimension to the concept of machine learning.^[8]

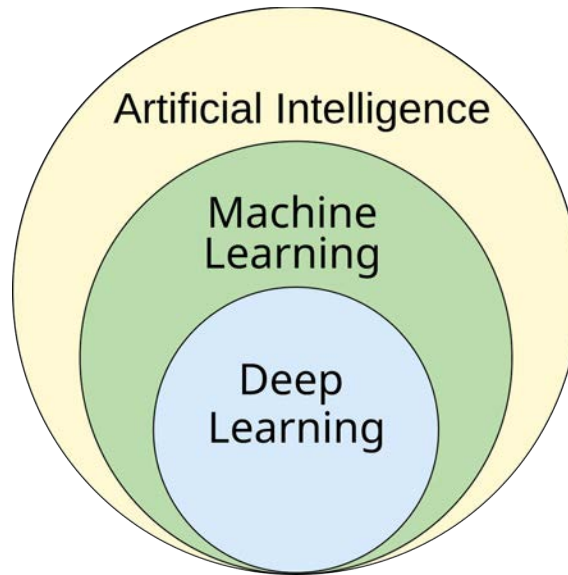


Figure 1: Machine Learning in relation to Artificial Intelligence^[9]

Machine Learning algorithms are primarily categorized in these three cases of learning, the Supervised Learning, the Unsupervised Learning and the Reinforcement Learning.

Supervised Learning

Supervised Learning stands out as the most widespread approach in the field of Machine Learning^{[10][41]}. At the core of this method lies the idea of learning an approximation function, where the algorithm is trained through examples of input-output data, i.e., pairs of (x,y) ^[10]. In this way, the goal is for the system, after being trained, to make a reliable prediction y^* for each unknown input sample x^* . Such algorithms find application in systems for tasks such as spam classification, facial recognition in images, as well as in medical diagnosis systems for patients^{[10][42]}.

The input data x in Supervised Learning problems can vary significantly, from simple value vectors to more complex structures such as texts, visual material, genomic sequences, or graphs^[10]. Correspondingly, the outputs y can also belong to different types. A fundamental subcategory that has been extensively researched is binary classification, where the prediction is a choice between two classes (e.g., spam or not-spam). Other subcategories include multiclass classification, in which the prediction y can be one of k classes, multilabel classification, in which the prediction y can be several of the k classes ranking problems, in some cases, the output variable y defines a partial order across a set, while in more general structured prediction tasks the output y may take the form of a combinatorial structure, such as a graph^[41]. A typical example of the latter is the automatic assignment of part-of-speech tags within a sentence. Finally, the prediction y can be a continuous real value.

The prediction process in these systems is based on a mapping $f(x)$ derived during training, which, for a given input x , produces the corresponding output y or a

probability distribution over the possible outputs^[41]. This mapping can take many different forms, ranging from decision trees and ensembles of trees to logistic regression, support vector machines, neural networks, kernel-based methods, and Bayesian models. A wide variety of algorithms have been proposed to learn such mappings, while more general frameworks, like boosting or multiple kernel learning, enable the combination of outputs from diverse learning approaches^[41].

To determine the function f during training, established principles from optimization theory and numerical analysis are often utilized, adapted to address the particular challenges of Machine Learning^[10]. The variety of training architectures creates the need to meet the different requirements set by its numerous applications. Specifically, each chosen algorithm offers distinct capabilities in capturing mathematical structures and in interpreting the extracted results, while simultaneously which results in different balances among how computationally intensive the method is, how much data is needed for training, and how well the model performs.^{[10][41]} As an illustration, Deep Neural Networks, with their multi-layered structure composed of simple parameterized computational units, are a characteristic example of the significant progress that has recently been achieved in the field of Supervised Learning.^{[10][42]}

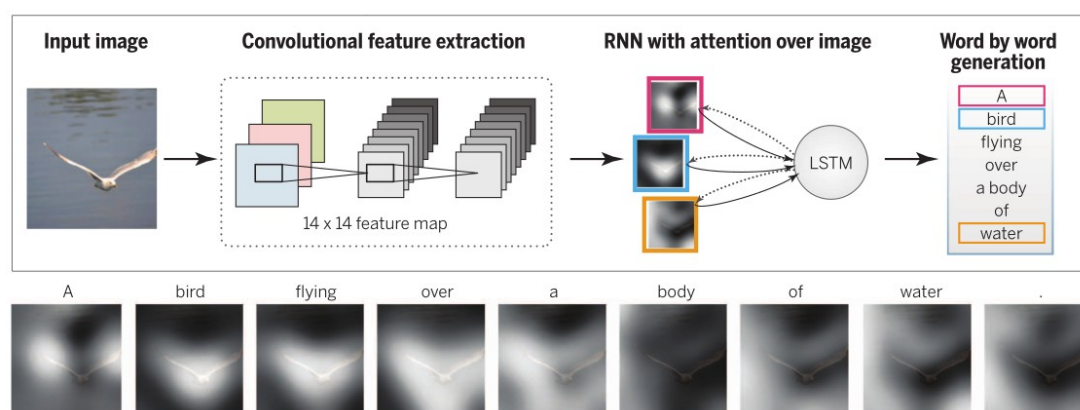


Figure 2: Automatic generation of text captions for images with Deep Networks.^[10]

Unsupervised Learning

A second main area in Machine Learning is Unsupervised Learning. In contrast to Supervised Learning, Unsupervised Learning is characterized by the analysis of data that does not have predefined labels^[10]. This process is based on the assumption that the data possesses inherent structural properties, which can be algebraic, combinatorial, or probabilistic^[10].

A characteristic approach in Unsupervised Learning is the assumption that the data, despite its apparent high dimensionality, actually lies on a low-dimensional manifold, the goal is therefore to identify this manifold directly from the data. To

achieve this purpose, various dimensionality reduction methods have been developed, such as Principal Components Analysis, manifold learning, factor analysis, random projections, and autoencoders^{[10][44]}. Each of these methods is based on specific assumptions regarding the geometry or structure of the underlying manifold. Topic modeling provides another approach to dimensionality reduction. To apply such methods, one first defines an appropriate objective function that reflects the underlying assumptions, frequently grounded in statistical frameworks such as maximum likelihood estimation, moment-based methods, or Bayesian inference. Optimization or sampling procedures are then employed to estimate the parameters that best satisfy this objective ^[44].

Another important application of Unsupervised Learning is clustering, which involves partitioning observed data into groups and creating a rule to predict the group for future data, without pre-existing labels to indicate a desired partition^[43]. Numerous clustering algorithms have been proposed, each based on distinct assumptions about the definition of a “cluster” ^[43]. In both clustering and dimensionality reduction, algorithmic efficiency is a central issue, since these methods are often applied to very large datasets where labels, essential in supervised learning, are not required.^{[10][43]}

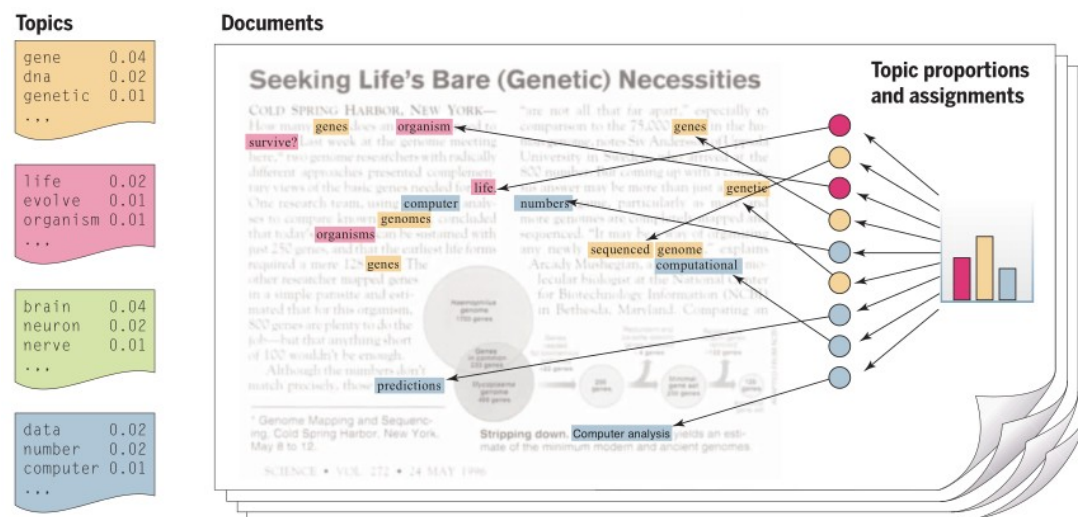


Figure 3: Example of topic modeling, a method used for document analysis.^[10]

Reinforcement Learning

Reinforcement Learning is the third main paradigm in the field of Machine Learning^[10]. This paradigm positions itself between Supervised and Unsupervised Learning in terms of the information provided by the training data. Rather than offering explicit input–output pairs, as in supervised settings, reinforcement learning supplies only feedback on whether a chosen action is appropriate. When an action is judged incorrect, the task of determining the proper alternative remains unresolved.^[10]

In sequential decision-making settings, reward signals are generally assumed to correspond to the entire sequence of actions rather than being attributed directly to specific steps ^[45]. Simplified scenarios, referred to as bandit problems, do provide feedback after each individual action, yet most reinforcement learning formulations are framed within control theory. Within this framework, the objective is to learn a control policy that guides an agent operating in an uncertain and dynamic environment, enabling it to choose actions in each state so as to maximize its long-term expected reward ^[45].

Reinforcement Learning is closely linked to the fields of control theory and operations research, where frameworks such as Markov decision processes (MDPs) and partially observable MDPs provide formal bridges. Many RL algorithms incorporate classical methods from control theory, including policy and value iteration, rollouts, and techniques for variance reduction ^{[10][45]}. In adapting these ideas to the context of Machine Learning, additional contributions have been introduced—such as approaches to handle large-scale problems, minimal assumptions regarding the unknown dynamics of the environment, and the use of supervised learning models to represent policies ^[10]. Furthermore, RL has significant interdisciplinary ties, particularly with psychology and neuroscience, where for instance RL-based models have been employed to describe dopamine neuron activity in monkeys learning to associate a visual cue with a sugar reward.^[10]

2.1.3 Neural Networks (NNs)

The main inspiration for the study of Neural Networks was the fact that the human brain performed computations in a very different way from how a conventional computer did.^[11] In the brain, its structural components, called neurons, are organized in an extremely complex, non-linear manner, giving it the ability to perform various computations extremely quickly, such as pattern recognition and perception. The brain achieves this because it has the ability from a very young age to build its own rules through experience.^[11]

Just like the human brain, a Neural Network is also composed of artificial neurons. Thus, a Neural Network is a machine that models the way in which the brain performs a function of interest. In order to be able to model such a function, the Neural Network must undergo the process of learning, after it has first been created through the interconnection of many artificial neurons. In this way, we can provide the following definition:

“A neural network can be viewed as a highly parallelized and distributed computational system, built from numerous simple units, with an inherent ability to capture experiential knowledge and reuse it when needed. Its operation is often compared to that of the human brain in two key ways:

1. The network acquires knowledge from its surroundings through a training process.
2. The learned information is encoded in the strengths of the interconnections between neurons, referred to as synaptic weights.”^[11]

The beginning of the history of Neural Networks is often identified with the pioneering work of Warren McCulloch and Walter Pitts titled "A Logical Calculus of the Ideas Immanent in Nervous Activity".^[11] They presented the first mathematical model of an artificial neuron, which operated as a simple logic gate, receiving binary inputs and producing a binary output, with its connection weights being predetermined. A few years later, in 1949, Donald Hebb introduced a fundamental learning principle, known as Hebbian Learning, which proposed that the strength of a connection between two neurons is reinforced each time that connection is used^[11].

The period from the late 1950s to the early 1960s was characterized as the "Golden Age" of early research in Neural Networks. In 1958, Frank Rosenblatt developed the Perceptron, an electronic device, and a learning rule (perceptron learning rule) that allowed it to learn^[11]. Two years later, in 1960, Bernard Widrow and Marcian Hoff introduced the ADALINE (Adaptive Linear Neuron) model, which used the Least Mean Squares (LMS) model for learning and found application in adaptive signal processing, and more specifically in adaptive noise filtering.^[11]

However, from the mid-1960s to the early 1980s, the field entered the first winter of Artificial Intelligence. A decisive role in this was played by the publication of the book "Perceptrons: An Introduction to Computational Geometry" by Marvin and Papert in 1969, which highlighted with mathematical rigor the limitations of simple single-layer Perceptrons, such as their inability to solve non-linearly separable problems like XOR. This critique led to a significant reduction in funding and research interest, although some researchers such as Grossberg, Klopff, and Fukushima continued their efforts^[11].

Today, the progress that has been made in the field of neural networks is remarkable, attracting significant interest and securing funding for further research. Chips based on neural theory are being developed, as well as applications for solving complex problems^[12]. In 2014, scientists from IBM introduced the TrueNorth processor, which featured an architecture similar to that of the brain. This IC had the capability to simulate in real time the operation of millions of neurons and 256 million connections, performing from 46 to 400 billion synaptic operations per second, and was the size of a postage stamp.^[12] The modern era of neural networks is also closely linked with the rise of deep learning, whose breakthrough applications in vision, speech, and natural language processing have been highlighted by LeCun, Bengio, and Hinton^[46].

The structure of a Neural Network is determined by the individual neurons, the way they are organized into layers, and their interconnection. These elements compose the architecture of the network, which allows it to process information and learn from data^[11].

Artificial Neuron / Unit / Node

The artificial neuron, commonly called a unit or node, is the basic building block of neural networks and is conceptually modeled after the behavior of biological neurons.^[11] Each neuron receives one or more input signals ($x_1, x_2, \dots, x_n$), which can be either the initial input values of the problem or the outputs of other neurons from previous layers. Each the inputs will be multiplied by a corresponding numerical coefficient called a weight ($w_1, w_2, \dots, w_n$), which corresponds to the strength of

each input. Subsequently, all the inputs, after being multiplied by their respective weights, are algebraically summed by an adder. After this summation, a bias (b) is usually added, which acts as a shift that gives the ability to the activation function to be moved to the left or to the right^{[11][46]}. The activation function $\phi(\cdot)$ is the final stage of the neuron, through which the total sum passes to produce the neuron's final output. The activation function is critical as it introduces non-linearity into the model, allowing the network to approximate complex, non-linear relationships in the data. Depending on the requirements of the problem and the architecture of the network, various activation functions exist^{[11][46]}.

Mathematically, the artificial neuron can be interpreted by the following equations:

$$u_k = \sum_{j=1}^m w_{kj} x_j$$

and

$$y_k = \phi(u_k + b_k)$$

where $x_1, x_2, \dots, x_m$ are the inputs of the neuron, $w_1, w_2, \dots, w_m$ are the corresponding weights of the inputs, u_k is the result of the summation, b_k is the bias, ϕ is the activation function and y_k is the output of the neuron^[11].

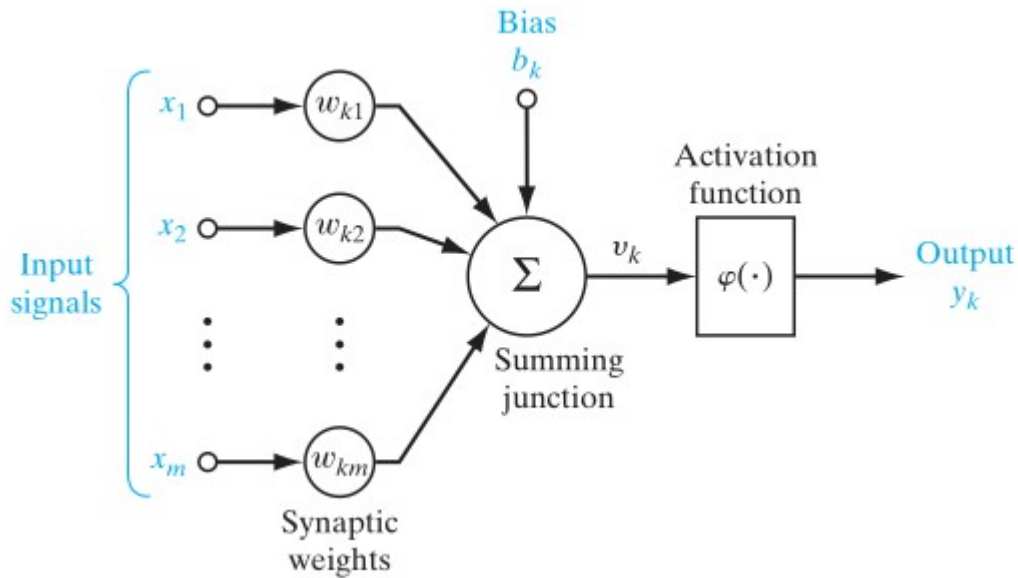


Figure 4: Non-linear model of an Artificial Neuron.^[11]

The activation function $\phi(\cdot)$ defines the output of the neuron for the activation potential v ^[11].

One of the first activation functions is the Threshold Function, which produces a binary output depending on v . Mathematically, this function is defined in the following way:

$$\varphi(v) = \begin{cases} 1 & \text{if } v \geq 0 \\ 0 & \text{if } v < 0 \end{cases}$$

However, for training networks with gradient-based methods, continuous and differentiable functions are preferred^[11]. The most usual category of these functions is the Sigmoid Functions, which have a characteristic "S" shape. One such function is the logistic function, which is mathematically defined as:

$$\varphi(v) = \frac{1}{1 + \exp(-av)}$$

In this case, the parameter a controls the steepness of the function, whose output values lie between 0 and 1.^[11]

Another Sigmoid Function is the hyperbolic tangent function, which is mathematically defined as:

$$\varphi(v) = \tanh(v)$$

and takes values from -1 to 1.^[11]

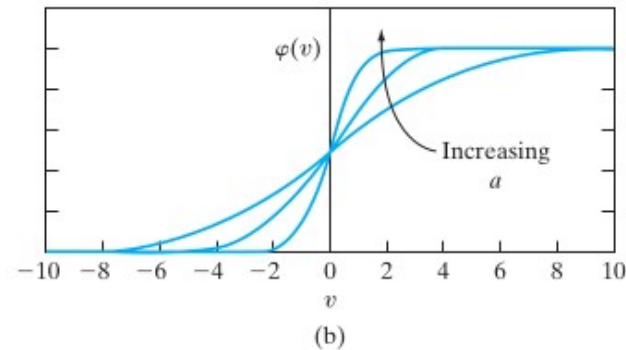
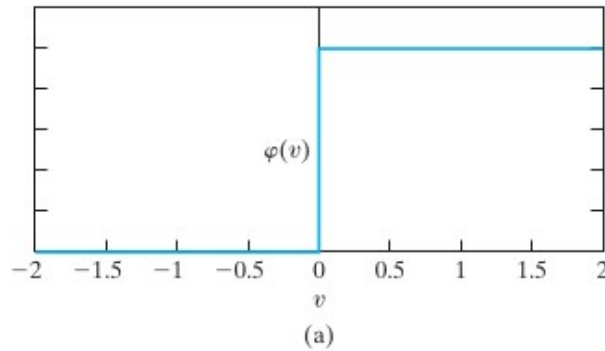


Figure 5: (a) Threshold Function (b) Sigmoid function for varying^[11]

Layers

Neurons in a Neural Network do not operate individually, but are organized into successive groups called layers. The flow of information usually occurs from one layer to the next. The main types of layers found in a Neural Network are the input layer, the hidden layers, and the output layer^{[11][46]}.

- **Input Layer:** This is the first layer of neurons, though they do not operate like classic neurons. Specifically, the purpose of this layer is to receive the initial data, i.e., the feature vectors that describe each problem sample, and to feed it to the next layer, which is either the first hidden layer or the output layer. Furthermore, no weights or activation function are applied at the input layer^[11].
- **Hidden Layers:** These are the layers located between the Input Layer and the Output Layer in a Neural Network. These layers consist of neurons that receive as input the outputs of the neurons from the previous layer. This can be either the Input Layer or the preceding Hidden Layer. The resulting outputs are passed forward either to subsequent hidden layers or directly to the output layer. Both the depth of the network (i.e., the number of hidden layers) and the number of neurons per layer represent key architectural choices. These Hidden Layers allow the network to extract intermediate features from the data, which are not directly visible in the initial inputs^{[11][46]}.
- **Output Layer:** This is the final layer of the network and it produces the final response or prediction of the model for a given input sample. The structure of the Output Layer, namely the number of neurons and the type of activation function used, depends directly on the type of problem the Neural Network is called upon to solve. For example, in a binary classification problem, the output layer may have a single neuron with a Sigmoid activation function, whereas in a multiclass classification problem, it may have as many neurons as there are classes, with a softmax activation function^{[11][46]}.

The visual representation of a neural network is often done using a directed graph, where the nodes of the graph correspond to the neurons (or the inputs) and the directed edges to the connections between them, which carry the corresponding weights.^[11]

Basic Network Architectures

One of the most fundamental and widely spread Neural Network architectures is that of Feedforward Neural Networks. The main characteristic of these networks is that information flows strictly in one direction: from the input layer, through the hidden layers (if they exist), to the output layer. No loops are formed in the connections between neurons, which means that the output of a neuron cannot affect its own input or the inputs of neurons in previous layers during the same pass of information processing.^{[11][46]}

The simplest form of such a network is the Single-Layer Feedforward Networks. These networks consist of only one Input Layer which is directly connected to an Output Layer, without any intermediate Hidden Layers. The term Single-Layer refers to the Output Layer, as the Input Layer is not counted because no computations are performed in it. The simplicity of these networks makes them capable of solving only linear problems^{[11][46]}.

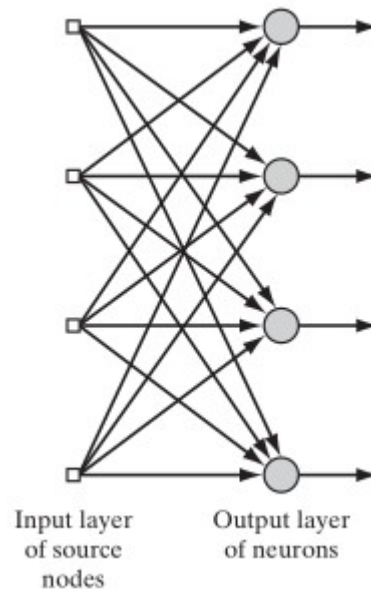


Figure 6: Feedforward Network with a Single Layer of neurons^[11]

Another architecture is Multilayer Feedforward Networks, that incorporate one or several hidden layers positioned between the input and output layers. In contrast to Single-Layer Feedforward Networks, the existence of these layers, in combination with the use of non-linear activation functions in their neurons, gives these networks the ability to approximate extremely complex non-linear functions and to model composite relationships in the data. Such networks are typically described by specifying the number of units in each layer. For instance, the model illustrated in Figure 7 can be denoted as a 10-4-2 network, indicating 10 input neurons, 4 neurons in a single hidden layer, and 2 output neurons^{[11][46]}.

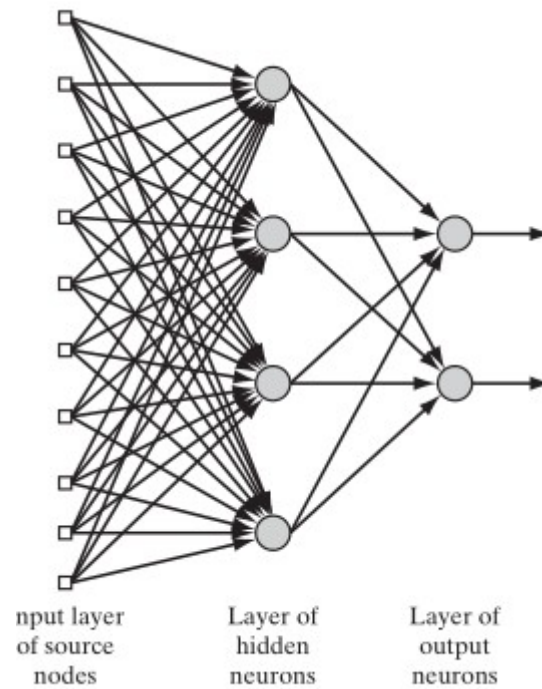


Figure 7: Fully connected Multilayer Feedforward Network with one Hidden Layer.^[11]

While Feedforward Networks constitute the fundamental category and the basis for understanding many concepts, other, more specialized architectures also exist. Two of the most important are Convolutional Neural Networks (CNNs), which are particularly effective in processing data with a spatial structure such as images, and Recurrent Neural Networks (RNNs), which are designed for processing sequential data, such as text or time series. These two architectures often incorporate or extend the basic principles of Feedforward networks^{[11][42][46]}.

2.2 Arrhythmia Detection with Neural Networks

According to the World health Organization cardiovascular diseases (CVDs) are the leading cause of mortality worldwide, accounting for an estimated 17.9 million deaths each year^[40]. Within this broad category, cardiac arrhythmias are a critical concern due to their potential severity. Arrhythmia can lead to serious complications and often manifest intermittently or without obvious symptoms, making timely detection essential for prevention of strokes, heart failure and sudden cardiac death.

Electrocardiography (ECG) is the standard method for monitoring heart activity and identifying arrhythmias, as it records the heart's electrical signals in a non-invasive manner. Despite its clinical value, conventional ECG interpretation requires significant effort and depends heavily on the expertise of trained professionals. A human specialist must comb through lengthy recordings for rhythm

irregularities, which is time consuming and may introduce variability. Moreover, brief ECG snapshots may miss sporadic arrhythmias that occur outside the recording window. Continuous long-term monitoring generates huge volumes of data that are impractical for manual review in a timely manner.^[13]

These challenges underscore the necessity for Artificial Intelligence (AI) techniques, particularly Neural Networks in modern cardiology. Deep Learning Neural Networks have the ability to automatically learn complex patterns in ECG signals and have demonstrated superior accuracy in detecting cardiac abnormalities compared to traditional rule-based learning methods.^[13] By leveraging large datasets of labeled ECG recordings, neural networks are trained to recognize the distinctive waveform patterns of various arrhythmias, often exceeding human performance. In fact, recent AI models achieved cardiologist-level accuracy in arrhythmia detection. One study reported an AI algorithm getting an F1 score of 0.84 versus 0.78 for cardiologists and higher sensitivity that experts across multiple rhythm classes. Such results highlight how neural networks can serve as a valuable assistive tool in ECG reading.^[14]

Beyond accuracy, neural network-based systems offer efficiency and consistency. Once deployed, an AI system can continuously monitor incoming ECG data in real-time and flag arrhythmic events instantly, enabling earlier intervention. Automation also reduces the workload on healthcare providers by selection normal from abnormal rhythms and minimizing human error in long duration monitoring. Given the rising demand for cardiac screening, Neural Networks fill an important gap by providing scalable, high speed analysis for early detection of arrhythmias and other cardiac conditions. In summary, the integration of Neural Networks into cardiology is proving essential for improving the accuracy, speed and accessibility of cardiac health prediction and diagnosis.^[14]

2.2.1 Recent Technological Approaches and Notable Studies

Modern approaches to cardiac health prediction heavily exploit deep learning, with Convolutional Neural Networks (CNNs) being the most used model. Researchers have also explored LSTMs, transformer-based models and various hybrid architectures. Next, we discuss how these Neural Networks architectures work and why they suit ECG data based on notable recent studies that apply them in cardiology.

2.2.1.1 Convolutional Neural Networks for ECG Analysis

Convolutional Neural Networks are a class of deep neural networks for pattern recognition in grid-like data^[13]. In ECG application, CNNs can identify patterns from the data like the shape and duration of heartbeats. There are two types of CNNs used for the ECGs data, 1-D CNNs and 2-D CNNs. A 1-D CNN slides small trainable filters (kernels) along the time axis of the ECG, automatically learning to recognize local waveform shapes. On the other side, a 2-D CNN turn the ECG data into images or other two dimensional formats and learn patterns form them. The filters used for the CNNs act as feature extractors, capturing clinically relevant attributes like the morphology and duration of heartbeats that are crucial for arrhythmia diagnosis. A

CNN by understanding characteristic wave pattern, can recognize the difference between normal and abnormal heart rhythms.^[13]

One of the biggest advantage of CNNs is that they don't need manual feature engineering. The network is able to learn the optimal features directly from the ECG data. This has led to improved arrhythmia classification accuracy over traditional methods, as evidenced by many studies.^[13] Bayani and Kargar, on their recent study "LDCNN: A new arrhythmia detection technique with ECG signals using a linear deep convolutional neural network" proposed a 1-D Linear Deep CNN (LDCNN) tailored for arrhythmia detection. Their LDCNN architecture uses multiple sequential convolutional layers, eight for the PTB dataset and five for MIT-BIH, and each of these layers is associated with a pooling layer. After the flattening layer, a dropout layer is subsequently applied to generate the final feature vector. This approach yielded excellent results, with **99.24%** and **99.38%** classification accuracy on the PTB Diagnostic ECG and MIT-BIH Arrhythmia databases. Such high performance illustrates the effectiveness of deep CNNs in learning discriminatory heartbeat features directly from raw ECG data.^[15]

Layer name	Filters	Kernel size	Activation function	Input shape	Output shape	Dropout rate	Pool size (Avg)	Padding	Stride
Input layer	-	-	-	1 × 360	-	-	-	-	-
Conv1D layer 1	16	13	ReLU	1 × 360	16 × 360	-	-	Same	1
AvgPool1D	-	-	-	16 × 360	179 × 16	-	3	-	2
Conv1D layer 2	32	15	ReLU	179 × 16	32 × 179	-	-	Same	1
AvgPool1D	-	-	-	32 × 179	32 × 89	-	3	-	2
Conv1D layer 3	64	17	ReLU	32 × 89	64 × 89	-	-	Same	1
AvgPool1D	-	-	-	64 × 89	64 × 44	-	3	-	2
Conv1D layer 4	128	19	ReLU	64 × 44	128 × 44	-	-	Same	1
AvgPool1D	-	-	-	128 × 44	128 × 21	-	3	-	2
Conv1D layer 5	256	21	ReLU	128 × 21	256 × 21	-	-	Same	1
AvgPool1D	-	-	-	256 × 21	256 × 10	-	3	-	2
Flatten layer	-	-	-	256 × 10	2560	-	-	-	-
Dropout layer	-	-	-	2560	2560	0.5	-	-	-
Dense layer 1	35	-	-	2560	35	-	-	-	-
Dense layer 2	5	-	-	35	5	-	-	-	-
Output layer	-	-	Softmax	5	5	-	-	-	-

Figure 8: The parameters used in configuring the LDCNN model for the MIT-BIH Arrhythmia dataset.^[15]

Layer name	Filters	Kernel size	Activation function	Input shape	Output shape	Dropout rate	Pool size (max)	Padding	Stride
Input layer	-	-	-	1×187	-	-	-	-	-
Conv1D layer 1	16	5	ReLU	1×187	16×183	-	-	Valid	1
Conv1D layer 2	16	5	ReLU	16×183	16×179	-	-	Valid	1
MaxPool1D	-	-	-	16×179	16×89	-	2	-	1
Dropout 1	-	-	-	16×89	16×89	0.1	-	-	-
Conv1D layer 3	32	3	ReLU	16×89	32×87	-	-	Valid	1
Conv1D layer 4	32	3	ReLU	32×87	32×85	-	-	Valid	1
MaxPool1D	-	-	-	32×85	32×42	-	2	-	1
Dropout 2	-	-	-	32×42	32×42	0.1	-	-	-
Conv1D layer 5	32	3	ReLU	32×42	32×40	-	-	Valid	1
Conv1D layer 6	32	3	ReLU	32×40	32×38	-	-	Valid	1
MaxPool1D	-	-	-	32×38	32×19	-	2	-	1
Dropout 3	-	-	-	32×19	32×19	0.1	-	-	-
Conv1D layer 7	256	3	ReLU	32×19	256×17	-	-	Valid	1
Conv1D layer 8	256	3	ReLU	256×17	256×15	-	-	Valid	1
GlobalMaxPool1D	-	-	-	256×15	256	-	-	-	1
Dropout 4	-	-	-	256	256	0.2	-	-	-
Dense layer 1	64	-	ReLU	256	64	-	-	-	-
Dense layer 2	64	-	ReLU	64	64	-	-	-	-
Output layer	1	-	Sigmoid	64	1	-	-	-	-

Figure 9: The parameters used in configuring the LDCNN model for the PTB diagnostic ECG dataset.^[15]

In addition to these CNN models developed and tested for cardiac arrhythmia detection, this thesis also utilized the arrhythmia-detection codebase by Cristiano Aguiar (2020).^[31] This open-source GitHub repository provides a pipeline for ECG signal analysis, specifically for the MIT-BIH Arrhythmia Database. Also it uses a 1D-CNN model which after training achieves 82% accuracy. Although, this model achieves 17% lower accuracy, the model has 17.321 parameters, much lower than the 950.000+ parameters used for the model which manage to achieve 99,38% accuracy.^[31]

2.2.1.2 Long Short-Term Memory Networks for ECG Analysis

Long Short-Term Memory Networks (LSTMs) are a form of RNNs, which incorporate gating mechanisms and an internal cell state that together regulate the flow of information and preserve long-term context^[13]. This architecture enables LSTMs to retain important signals over extended time steps while avoiding gradient vanishing, which is particularly useful for ECG analysis where an abnormal rhythm might develop over many seconds or minutes.^[13]

A LSTM model was used by Muhammad Ashfaq Khan and Yangwoo Kim in their study "Cardiac Arrhythmia Disease Classification Using LSTM Deep Learning Approach" to classify cardiac arrhythmias into 16 distinct categories using ECG data.

Their approach utilized the UCI Arrhythmia Dataset, which consists of 279 attributes, including patient demographics, ECG features, and other medical data. Then they use it to train a LSTM model, chosen for its ability to capture long-range dependencies in sequential data, making it ideal for ECG signal analysis where the temporal order of heartbeats is critical. To enhance performance, they first applied Principal Component Analysis (PCA) for noise removal, a common technique in signal processing that helps eliminate irrelevant features and improve the model's training speed and accuracy. The authors reported a **93.5%** classification accuracy, significantly outperforming traditional machine learning classifiers like SVM and Random Forests, which achieved much lower accuracy on the same dataset.^[16]

DL approach	Accuracy (%)	Precision (%)	Recall (%)	F1-score (%)
LSTM	91.37	90.0	90.2	90.09
PCA + LSTM	93.5	92.8	90.7	91.73

Figure 10: Performance metrics of deep learning models.^[16]

Method	Accuracy (%)
VF15 based arrhythmia classification approach [27]	62.0
Classifier comparison [20]	66.9
Different ML approaches [23]	72.7
KNN [35]	88.2
SVM based approach [31,32]	73.8
ISSA algorithm [63]	84.22
AIRS with fuzzy [64]	80.77
Ensemble ML [65]	80.0
SVM and NB ML [66]	89.7
DL approach proposed approach	93.5

Figure 11: Comparison of LSTM DL model with previous models using cardiac arrhythmia dataset.^[16]

2.2.1.3 Hybrid and Advanced Neural Network Architectures

In practice, the highest-performing cardiac prediction models often combine multiple neural network types to harness their complementary strengths.^[17] Hybrid architectures have become common in recent literature. For example, Liu H. W. developed a combined CNN+RNN classifier for ECG analysis and found that adding a recurrent layer boosted sensitivity by about **28%** compared to using a CNN alone. In another comparative experiment, a CNN by itself achieved about **93.6%** accuracy, but integrating an LSTM improved accuracy to **95.1%**, and a another model using convolutional LSTM reached **97.6%** on the same arrhythmia classification task.^[17]

This approach is used in the article “DenseNet-ABiLSTM: Revolutionizing Multiclass Arrhythmia Detection and Classification Using Hybrid Deep Learning Approach Leveraging PPG Signals”. This study introduce a hybrid model called DenseNet-ABiLSTM, which integrates a densely-connected CNN (DenseNet) with an attention-based bidirectional LSTM for multiclass arrhythmia detection. The CNN component learns multi-scale features from the input and the Bi-LSTM captures the sequence of those features. An attention mechanism highlights the most informative time steps for the final classification. This hybrid model classified six types of arrhythmias with a mean F1-score of $\sim 87.7\%$ and accuracy $\sim 89.1\%$, outperforming many single-model approaches.^[18]

Another emerging trend is combining CNNs with Transformers. Transformers are a newer architecture originally developed for natural language processing, now gaining traction in time-series tasks including ECG analysis. The reason is attention mechanism, which allows the model to weigh the importance of different time points in the sequence when making a prediction. Transformer architectures process ECG signals as ordered sequences of data points, enabling them to focus on salient features while modeling temporal dependencies across the entire input. As a result, they can effectively capture both local and long-range patterns.^[13]

A hybrid CNN-transformer model was used in the study called “A novel hybrid CNN-transformer model for arrhythmia detection without R-peak identification using stockwell transform”. This model uses the stockwell transform (S-transform) for time-frequency feature extraction. The CNN model capture local patterns while the transformer architecture learns long-term dependencies. With this model the authors manage to achieve accuracy **99.58%** on the MIT-BIH dataset and **97.8%** on the Icential1k dataset. One of the reasons that this model can achieve such accuracy is the fact that this method has the advantage to not require R-peak detection, which can fail in certain arrhythmias. So by not relying on R-peak the model demonstrated greater accuracy, particularly in arrhythmias where R-peak identification is challenging.^[19]

Methods	Acc(%)	N			S			V			Q			F		
		F1	Se	PPV	F1	Se	PPV	F1	Se	PPV	F1	Se	PPV	F1	Se	PPV
Lin et al. ³³	99.2	.962	.973	.951	.908	.905	.911	.982	.985	.979	–	–	–	.991	.983	1.00
Oliveira et al. ³⁴	95.3	.974	.971	.978	.916	.761	.566	.950	.930	.950	–	–	–	–	–	–
Chen et al. ³⁵	93.1	.969	.984	.954	.334	.295	.384	.773	.708	.851	–	–	–	–	–	–
Kung et al. ³⁶	98.6	–	–	–	.815	.754	.887	.970	.967	.974	–	–	–	–	–	–
Ince et al. ³⁷	98.3	–	–	–	.581	.635	.537	.860	.846	.874	–	–	–	–	–	–
Shi et al. ³⁸	94.2	.971	.953	.989	.627	.907	.479	.885	.929	.845	–	–	–	–	–	–
Xie et al. ³⁹	96.5	.900	.943	.877	.848	.797	.906	.966	.971	.962	–	–	–	.816	.908	.741
Zhai et al. ⁴⁰	96.1	.899	.879	.920	.754	.768	.740	.931	.938	.924	–	–	–	.700	.624	.796
Feiyan et al. ³²	99.6	.999	.999	.999	.973	.980	.966	.990	.991	.989	–	–	–	.924	.907	.942
Proposed	99.6	.997	.988	.991	.997	.997	.997	.993	.994	.991	1.00	1.00	1.00	1.00	1.00	1.00

Figure 12: The performance of the proposed method compared to previous methods on the MIT-BIH dataset.^[19]

A hybrid architecture is also utilized by Cristiano Aguiar (2020), whose codebase used for this thesis as previous mentioned.^[31] Cristiano for his second

architecture uses a combination of 1D-CNN with LSTM. After training and evaluating, this model manage to achieve 80% accuracy, while having 17.330 parameters.^[31]

In summary, the deep learning toolkit for cardiac health prediction is rich and rapidly evolving. Convolutional Neural Networks provide a strong foundation for feature extraction from ECGs, LSTM networks add memory for temporal patterns, and transformers contribute advanced sequence modeling capabilities. By intelligently combining these, recent works have achieved unprecedented accuracy in predicting arrhythmias, often exceeding human expert performance. The following section will consider the remaining challenges faced by these approaches and how future research is directing efforts to make neural network solutions more robust and clinically useful.

2.2.2 Resource Barriers to Real-World ECG-AI

Despite the promising accuracy of neural networks in cardiac arrhythmia detection, their integration into widespread clinical and everyday use faces significant computational and resource constraints. These limitations hinder deployment in resource-limited settings (e.g., portable monitors, embedded medical devices) and exacerbate operational costs, latency, and energy consumption. This section synthesizes the primary challenges, emphasizing hardware and optimization barriers.

A primary challenge is the intensive computational and hardware requirements of modern AI models. State-of-the-art neural networks demand substantial processing power and memory to attain high accuracy, which typically confines their use to high-end servers or devices. In particular, many deep learning models rely on specialized hardware such as GPUs or AI-specific chips that consume considerable power and memory resources. Deploying full-fledged neural networks on a mobile or wearable device is often infeasible because typical these devices lack the computing power and storage capacity for these heavy models.^[20]

Another critical limitation involves power and energy constraints. Portable medical devices and wearables run on batteries and operate under strict energy budgets, but the deep neural computations are notoriously energy-hungry. In fact, Sazonov and Daoud in their article “Grand Challenges in Wearable Electronics”, they report that “Availability of power is the single, most important limiting factor in modern wearables”.^[21] In the article “Advancements in TinyML: Applications, Limitations, and Impact on IoT Devices”, it is also said that power consumption is a foremost concern, when it comes to incorporate MLs in portable devices. Small devices simply cannot run complex models continuously without quickly draining their batteries. This has forced developers to employ aggressive optimizations like low-bit quantization or pruning of networks to cut down on energy usage.^[20]

Memory and storage limitations further complicate deployment. Advanced deep learning models often consist of millions of parameters, requiring far more memory than typical embedded devices have available. Researchers have developed compact architectures that significantly reduce model size and computation at some cost to accuracy. Such devices may experience higher latency so they might not be able to meet real-time responsiveness requirements. In order to resolve the problem off limited memory, it has been tried the use off cloud by sending the data to it. That

workaround introduces its own challenges: reliance on cloud connectivity and added network delay, which is undesirable for time-sensitive clinical applications.

Finally, there is the challenge of translating AI software into practical hardware implementations. There is shortage of standardized tools and frameworks for developing MLs with the purpose of deploying them in resource-constrained devices, making really hard the research about it. Also, implementing advanced neural networks in a resource-constrained environment demands proficiency in the areas of machine learning and embedded systems, an intersection of skills that is still relatively rare.^[20]

To further highlight the challenges faced by existing models, two of the models mentioned in Section 2.2.1 achieved impressive performance, with 99.24% and 99.38% accuracy for the PTB Diagnostic ECG and MIT-BIH Arrhythmia databases, respectively. The first model had more than 250.000 parameters, while the second one had over 950.000 parameters^[15]. In comparison, the largest model used for this thesis had 8.721 parameters and manage to achieve 83,25% accuracy. However, converting this model to HDL using HLS4ML was both time and memory demanding process. Furthermore, the model demanded more than 120.000 LUTs and more than 119.000 FFs. Given that the two previously mentioned models are more than 20 times larger in terms of parameters, their deployment in conventional FPGA systems may not be feasible due to resource constraints. These large models would likely exceed the capacity of typical FPGAs.

Together, these factors form a set of substantial challenges. They currently inhibit the seamless adoption of this technology in widespread clinical practice and everyday health devices, even as research prototypes demonstrate what might be possible under ideal conditions. Each constraint will need to be carefully addressed through continued innovation in model optimization, hardware design, and system architecture before the technology can fully realize its potential at scale.

2.3 Neural Networks Size Reduction Techniques

Modern Deep Neural Networks have manage to achieve state-of-the-art accuracy by growing extremely large. This large size comes with the need of massive computational and memory demands making their deployment on devices with limited hardware impossible. To be able to have models for arrhythmia detection in everyday use this problem must be solved. Studies have shown that large Networks contain significant redundancy, so a variety of model compression techniques have been developed to reduce this redundancy. The leading strategies for reducing the size of neural networks include quantization, pruning, low-rank factorization, and knowledge distillation. The subsequent sections outline each of these techniques in brief.^[22]

2.3.1 Quantization

Quantization reduces the memory and computational cost of a Neural Networks by representing weights and activations with lower precision numerical format. This means converting high-precision formats, 32-bit floating point values (FP32), into fixed-point integers with fewer bits like 8-bit, 4-bit or even 2-bit values. By doing so, the storage size of each parameter shrinks dramatically. For example, 8-bit integers

use 1 byte instead of 4 bytes for FP32. This means that the memory usage is cut down by 75%. More extreme quantization (such as 4-bit or 2-bit) can yield compression ratios of 8× or 16× per parameter. In addition to memory savings, quantization often allows **faster inference** because arithmetic on low-bit integers can be executed more efficiently on modern hardware (utilizing bit-shift operations and integer ALUs).^[22]

The conversion from floating-point to fixed-point format is mathematically described with the equations:

$$q = \text{round}\left(\frac{r}{S} + Z\right) \quad (1)$$

$$S = \frac{r_{\max} - r_{\min}}{q_{\max} - q_{\min}} \quad (2)$$

$$Z = \text{round}\left(q_{\max} - \frac{r_{\max}}{S}\right) \quad (3)$$

where:

- r represents the floating-point value to be quantized.
- q is the quantized fixed-point integer.
- S is the scaling factor, determining the range mapping from floating-point to fixed-point values.
- Z denotes the zero-point, which guarantees that the integer encoding correctly corresponds to zero in the floating-point domain.
- $r_{\max}$ and $r_{\min}$ are the maximum and minimum floating-point values respectively, whereas $q_{\max}$ and $q_{\min}$ are their corresponding fixed-point bounds.

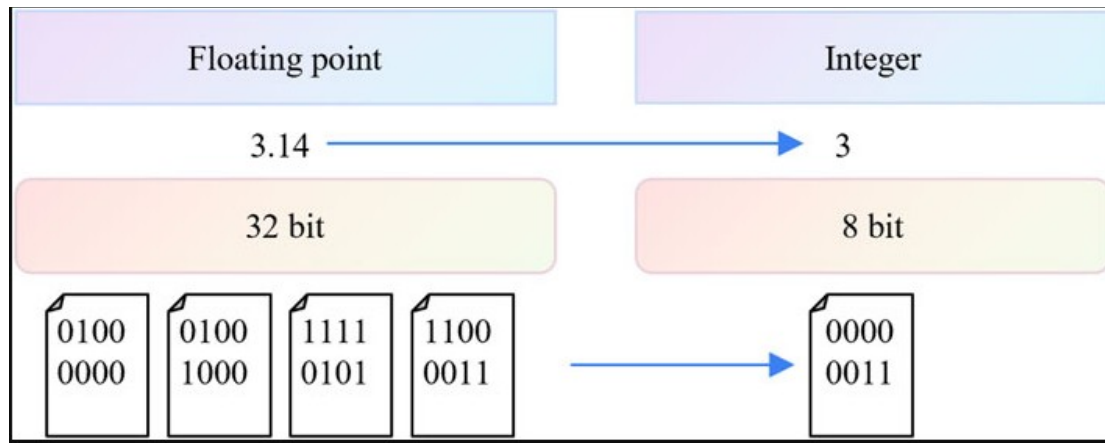


Figure 13: The process of model quantization^[22].

Quantization approaches are broadly categorized based on various criteria:

- **Symmetric and Asymmetric Quantization:** In symmetric quantization, the quantization intervals are symmetric around zero, whereas asymmetric quantization intervals are offset from zero.
- **Quantization Granularity:** The methods differ in their level of granularity, ranging from layer-wise to channel-wise and group-wise approaches. While finer granularity typically yields better performance, it demands additional computational resources and storage for parameters like scaling factors.
- **Post-Training Quantization (PTQ) and Quantization-Aware Training (QAT):** PTQ directly quantize pretrained models without additional training, relying on calibration to reduce errors. Conversely, QAT simulates quantization effects during training, fine-tuning the model to adapt to quantization-induced errors. QAT generally achieves higher accuracy but requires significant computational resources for retraining.
- **Quantization of Weights Only vs. Joint Quantization of Weights and Activations:** Weight-only quantization compresses only the weights while maintaining activation precision, preserving inference quality but offering limited computational acceleration. In contrast, joint quantization of weights and activations significantly lowers both memory and computational costs but may increase quantization error and reduce accuracy.
- **Simulated and Integer-only Quantization:** Simulated quantization uses lower precision solely for model weights and dequantize them to higher precision during inference, whereas integer-only quantization executes every operation using purely integer arithmetic, offering greater computational efficiency due to hardware optimization for integer operations.

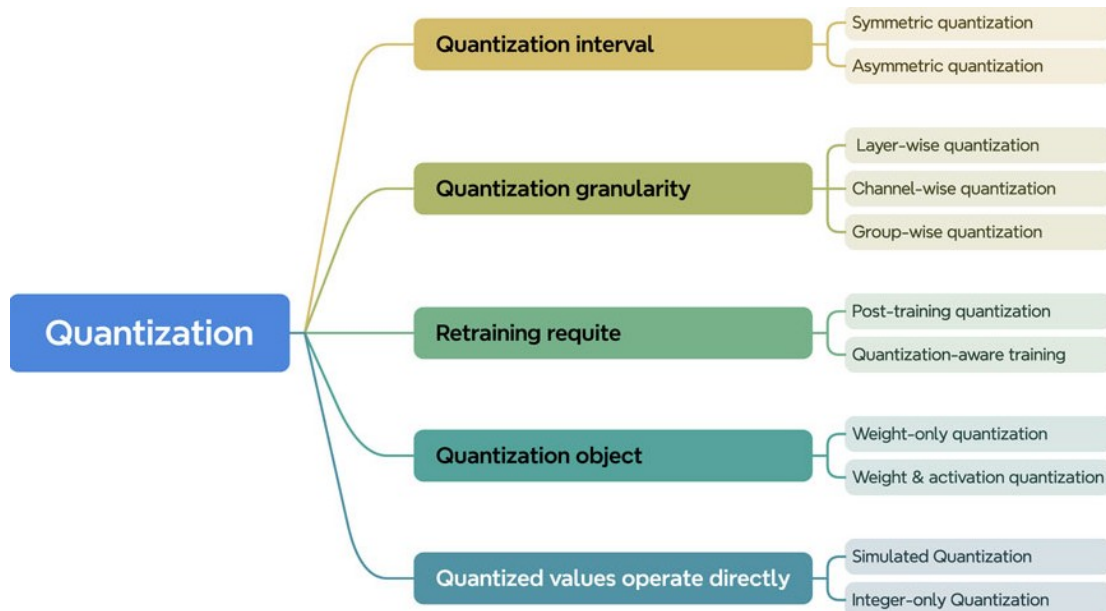


Figure 14: A taxonomy of quantization based on the focus of each method.^[22]

2.3.2 Pruning

Pruning techniques reduce a network's size by removing parts of the model that are considered unnecessary or redundant. The core idea is that not all weights or neurons are equally important for making correct predictions. Many parameters can be zeroed out or deleted with only a minor impact on accuracy. Pruning algorithms identify these insignificant parameters (for example, weights with near-zero values or neurons with minimal activation contribution) and eliminate them from the model, resulting in a sparser network. There are two broad categories of pruning: unstructured pruning and structured pruning.^[22]

2.3.2.1 Unstructured Pruning

In unstructured pruning, individual connections are removed independently by setting a fraction of the smallest-magnitude weights in the weight matrices to zero. Various pruning strategies focus on eliminating weights deemed insignificant.^[22] Han et al. (2015) introduced a prominent approach that prunes weights below a certain magnitude threshold and then retrain the pruned network using the original parameters to improve performance. However, such pruning results in an irregular, sparse matrix structure, requiring specialized hardware or software for efficient computation.^[23]

Mocanu et al. (2018), drawing inspiration from the natural sparsity observed in biological neural networks, proposed substituting fully connected layers with sparse topologies generated via Erdős–Rényi random graphs. This approach drastically decreases the number of trainable parameters while preserving predictive accuracy. During training, weights with the smallest magnitudes are pruned and replaced with new connections in a proportional manner, thus keeping the sparsity level constant. Although the method succeeds in discovering sparse architectures, it

incurs higher training costs and poses difficulties at inference time due to irregular connectivity, resulting in inefficient memory usage and poor cache performance.^[24]

2.3.2.2 Structured Pruning

Structured pruning, or unit pruning, removes entire neurons or convolutional filters along with all their connections. Structured pruning removes entire neurons along with their connections, which frequently results in lower performance compared to unstructured, weight-based pruning techniques.^[22]

Alqahtani et al. (2021) introduced a method that uses a voting mechanism to evaluate neuron importance by analyzing their activation levels. Neurons that contribute less significantly according to these scores are pruned, simplifying the network. Notably, this approach operates during the training phase itself, avoiding additional steps like pretraining or subsequent fine-tuning.^[25] In another strategy, Srinivas and Babu (2015) suggested pruning based on weight similarity within individual layers, removing neurons with comparable weights.^[26] Furthermore, Mariet and Sra (2016) introduced the DivNet framework, which applies a probabilistic methodology to detect neurons with comparable activation behaviors and consolidate them.^[27]

Numerous structured pruning techniques have been designed specifically for convolutional networks. For instance, Li et al. (2017) proposed ranking convolutional kernels according to the sum of their absolute weight values and eliminating those with the smallest scores.^[28] Similarly, Hu et al. (2016) proposed a data-driven method called Average Percentage of Zeros (APoZ) to assess the significance of neurons based on their activation sparsity.^[29]

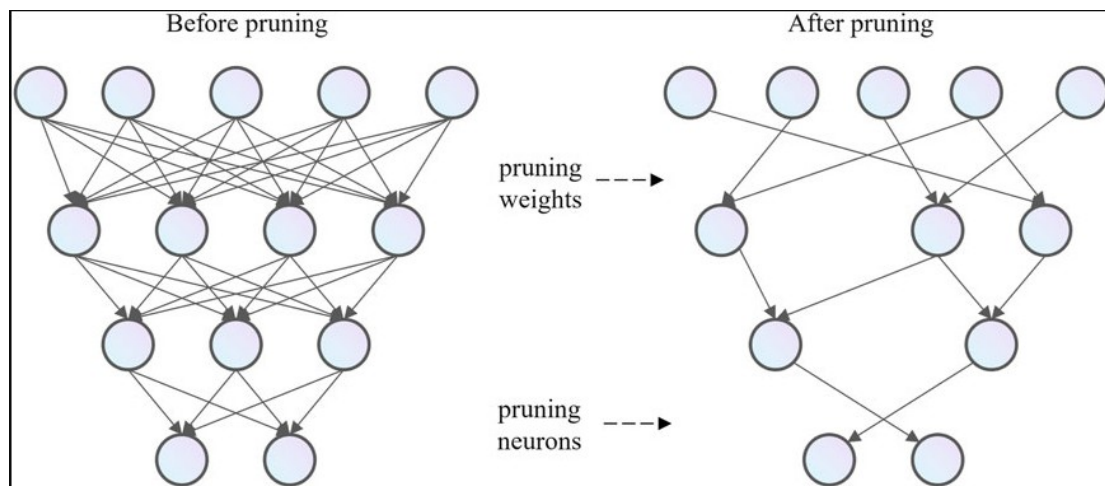


Figure 15: The structure of a neural network after pruning.^[22]

2.3.3 Low-Rank Factorization

Low-rank factorization is a technique that compresses neural networks by exploiting the linear algebraic redundancy in weight matrices or tensors. The principle is to approximate a large weight matrix by the product of two or more smaller matrices,

thereby reducing the number of parameters needed to represent the original operation. Within deep convolutional architectures, convolution operations account for the majority of computational cost; thus, compressing these layers can substantially improve both speedup and overall compression efficiency.^[22]

Many weight matrices in deep networks have a lot of redundant information. By performing a matrix factorization, we can find a lower-dimensional basis that captures most of the variability of the original weights, and then re-express the layer's computation in this compressed form.^[22] For example, a large 2D weight matrix of shape $m \times n$ can be factorized into an $m \times k$ matrix times a $k \times n$ matrix, with k chosen such that $k \ll \min(m, n)$, achieving substantial reduction in parameters.

In the context of Convolutional Neural Networks convolution operations are the most computationally demanding. With the compression of the convolutional layers, the acceleration ratio and the compression rate can be significantly improved. A convolutional kernel with rank R , can be broken into R separable convolutional kernels. A classic illustration is replacing a 7×7 convolutional layer with two layers: a 7×1 convolution followed by a 1×7 convolution, decreasing the number of parameters from 49 down to 14.^[22]

However, low-rank compression comes with some limitations. One challenge is that finding the optimal low-rank approximation for each layer can itself be computationally intensive, making it prohibited for large models. Moreover, simply factorizing a pretrained weight matrix and using the factors can lead to an accuracy drop. Although these techniques reduce the number of parameters, their performance is not always on par with that of alternative approaches.

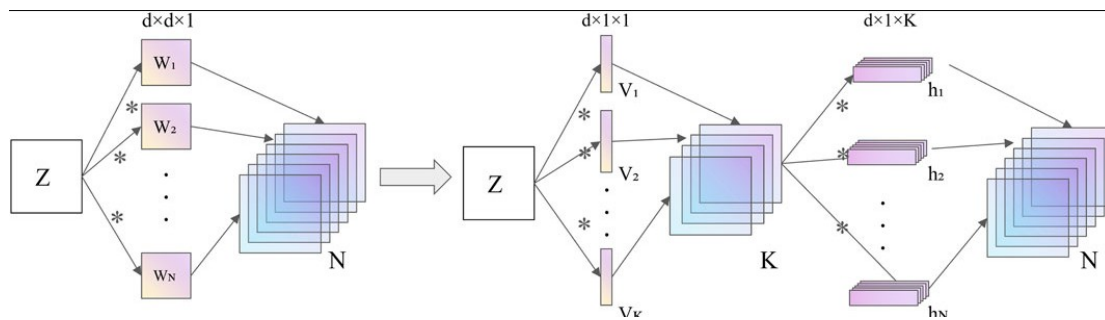


Figure 16: The core concept of low-rank decomposition.^[22]

2.3.4 Knowledge Distillation

Knowledge Distillation is a technique in which a large, high-performing neural network (the teacher) is used to guide the training of a much smaller network (the student) so that the student can achieve nearly the teacher's accuracy with far fewer parameters. Unlike quantization or pruning, which directly modify a given model's parameters, distillation creates a new model with a compact architecture and transfers knowledge to it by leveraging the teacher model's output. Two main forms of knowledge distillation exist: black-box and white-box. In black-box distillation, the student model relies solely on the teacher's output predictions, whereas white-box distillation leverages both the predictions and access to the teacher's parameters.

Since white-box methods exploit internal information about the teacher's structure and knowledge representation, they generally achieve superior performance.^[22]

Knowledge Distillation typically used to compress large language models (LLMs) and manage to achieve great results in compression of large models. Despite its strengths, knowledge distillation also has some challenges. The distillation process requires training the student, which can be time-consuming when the teacher is very large which also means it has significant computational cost. Moreover, the student's success heavily depends on the teacher's quality, a poorly performing teacher will transfer misleading knowledge. There is also the issue that some fine-grained information in the teacher might not be captured by the student, which might lower the student's performance in specialized tasks. All these problems raise the need of well thoughtful designed in order of this method to work.^[22]

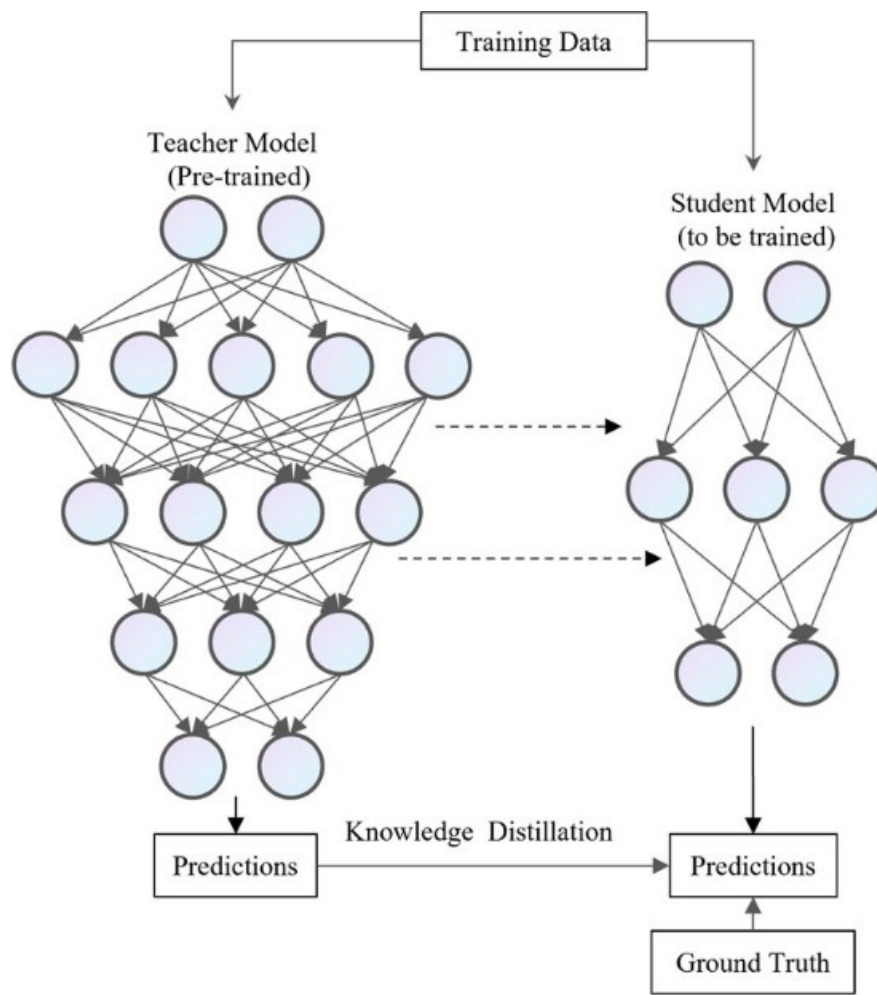


Figure 17: The process of model distillation.^[22]

Chapter 3: Neural Network Hardware Implementation Flow

Neural Networks can be very complex and thus their implementation on hardware may be difficult, very costly and time consuming. Therefore in recent years various tool flows have been proposed for the implementation of NN models while considering various algorithm and hardware design options. One popular tool flow is the so-called HLS4ML tool flow. HLS4ML is an open-source tool written in Python that enables the swift conversion of machine learning models, originally built in popular frameworks like Keras or PyTorch, into high-level synthesis (HLS) code that is compatible with FPGA implementations.^[30] The resulting HLS project can then be processed further to generate either a reusable IP core or a hardware kernel for CPU co-processing. A significant advantage of HLS4ML is its high degree of configurability, which allows users to adjust key parameters like quantization, parallelization, and latency/resource trade-offs to meet the specific needs of their applications.^[30]

The core inspiration for developing HLS4ML comes from the challenges faced in physics with very high energy, especially within the CERN Large Hadron Collider, where real-time data processing is crucial. Traditional machine learning approaches are typically applied for offline analysis, but the sheer volume of collision data at the LHC demands real-time decision-making to filter and retain relevant events. Using "trigger" systems, the LHC can efficiently filter events for further analysis. The deployment of neural networks directly onto FPGAs allows for much lower latency at the detector stage, making it possible to capture important events that could indicate new physical phenomena.^[30]

Beyond high-energy physics, FPGA implementations have been effectively utilized in the medical field for real-time detection of cardiovascular diseases (CVDs). For instance, a study by Chowdhury et al. (2024) demonstrated an FPGA-based system that classifies various CVDs using photoplethysmogram (PPG) signals. The system achieved high accuracy rates, including 93.48% for cerebral infarction and 96.43% for cerebrovascular disease, with a power consumption of just 0.693 W, highlighting the efficiency of FPGA implementations in medical diagnostics.^[49] Additionally, Liu et al. (2025) developed a lightweight convolutional neural network (CNN) accelerator on an FPGA platform for real-time arrhythmia classification, achieving a classification accuracy of 96.55% with reduced resource consumption and latency, further emphasizing the potential of FPGA-based solutions in healthcare applications.^[50]

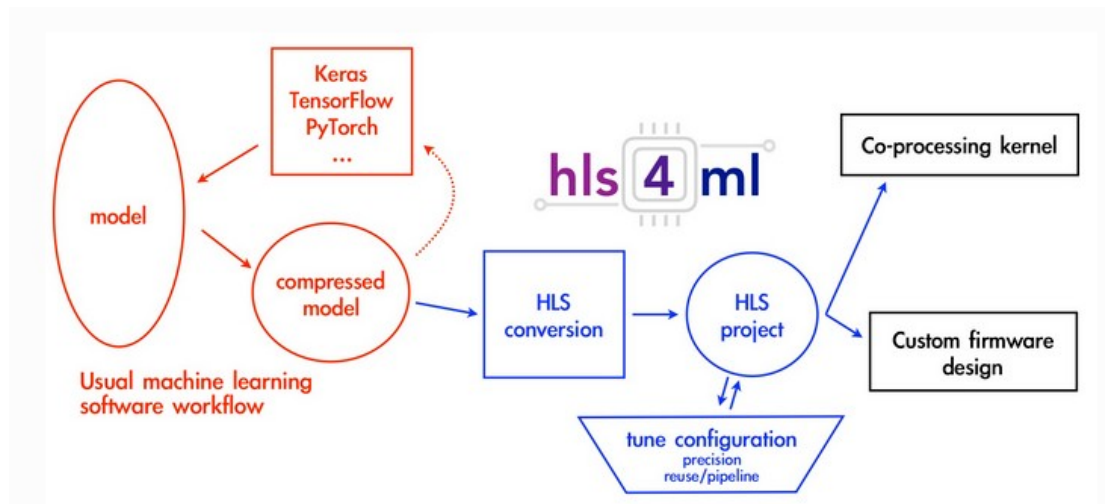


Figure 18: Overview of the HLS4ML workflow, converting trained machine learning models into optimized hardware designs for FPGA deployment.^[30]

3.1 Capabilities

The HLS4ML library integrates with several machine-learning ecosystems and supports a variety of network types and hardware compilers. A condensed overview is presented below.^[30]

Machine Learning Interfaces:

- Standard Keras and its quantization-aware variant QKeras
- PyTorch models through dedicated parsers
- ONNX and its quantized extension QONNX

Supported Neural Network Designs:

- Feed-forward multilayer perceptrons
- Convolutional models for image or spatial data
- Recurrent networks such as LSTM
- Graph-based topologies (e.g., GarNet)

Hardware Synthesis Backends:

- Xilinx Vitis HLS
- Xilinx Vivado HLS
- Intel FPGA HLS
- Mentor Catapult HLS
- Experimental support for oneAPI

3.2 Core Principles

At its core, a neural network layer computes the activation of each node from the weighted sum of the outputs of the previous layer plus a bias term. This result is then passed through a nonlinear activation function. In HLS4ML, such computations are arranged sequentially layer by layer, while pipeline scheduling allows new data to enter before the full computation of a layer has finished. Complex activations are often pre-tabulated and stored for efficiency.

To adapt a model to the FPGA fabric, the developer can adjust several design aspects:

- **Model Simplification & Compression** – reducing layer sizes or pruning weights lowers resource usage and speeds up compilation.
- **Arithmetic Precision** – users specify custom bit-widths for weights, activations, and accumulators.
- **Parallelization vs. Resource Reuse** – the trade-off between duplicating hardware units for high throughput and reusing them to minimize FPGA footprint.
- **Quantization-Aware Training** – training with reduced precision (e.g., via QKeras) improves inference accuracy in fixed-point hardware deployment.

A key performance enabler is the adoption of fixed-point arithmetic, which offers lower latency and reduced hardware cost compared to floating-point. Additionally, bit-width of lookup tables for activations can be customized.

The reuse factor parameter provides direct control over the balance between latency and hardware resources:

- A low reuse factor dedicates many multipliers to achieve minimum delay but consumes large amounts of logic.
- A high reuse factor shares multipliers across operations, conserving resources but extending execution time.
-

Frontends and Backends

HLS4ML separates frontends (responsible for parsing a trained ML model into an internal representation) from backends (responsible for generating hardware code). Example frontends are the Keras and ONNX parsers, while common backends include Vivado HLS, Intel HLS, and Vitis HLS.

Input/Output Transfer Modes

Two I/O schemes are available for moving data between layers:

- **Parallel I/O** – All elements of an array are transferred simultaneously. This mode maximizes parallelism, ideal for compact MLPs or small CNNs, but can quickly exceed FPGA resources.
- **Streaming I/O** – Data elements (e.g., image pixels) are fed sequentially, while channel information is grouped in parallel. Streaming is the recommended option for deep CNNs and recurrent networks. Buffers (FIFO) are inserted

automatically, which may overuse memory resources unless depth optimization is applied.

Computation Strategies

The term strategy denotes how the core matrix–vector multiplication is implemented. Depending on the backend, options range from latency-oriented (faster execution) to resource-efficient designs, with specialized methods available for particular layer types.

3.3 Internal

Internal representation

The HLS4ML library will parse models from Keras, PyTorch or ONNX into an internal execution graph.^[30] This model graph is represented with the `ModelGraph` class. The nodes in this graph, loosely corresponding to the layers and operations of the input model are represented by classes derived from the layer base class.^[30]

Layers are required to have defined inputs and outputs that define how they are connected in the graph and what is the shape of their output. All information about the layer's state and configuration is stored in its attributes. All weights, variables and data types are attributes and there are mapping views to sort through them. Layers can define expected attributes and can be verified for correctness, or to produce a list of configurable attributes that user can tweak.^[30]

Layers

Dense Layers

One-dimensional Dense Layers

Dense layers over one-dimensional data perform a matrix-vector multiplication followed by elementwise addition of bias tensor. This routine is the underlying computation of many other layers as well and is reused as much as possible. It exists in several implementations across different backends, for different `io_type`'s and strategies.^[30]

`io_parallel`

All the backends have a `Resource` implementation, which divides the computation into a loop of `reuse_factor` iterations, each iteration simultaneously accessing a different part of the array partitioned in BRAM. There are different implementations depending on whether the reuse factor is smaller or bigger than the input size. The two Xilinx backends and Catapult also implement a `Latency` implementation, which uses the reuse factor to control the amount of pipelining/unrolling of the whole function while the weight array is fully partitioned in registers.^[30]

`io_stream`

The `io_stream` implementation only wraps the `io_parallel` implementation with streams or pipes for communication. Internally, data is still accessed in parallel as an array.^[30]

Multi-dimensional Dense Layers

Multi-dimensional Dense layers are converted to pointwise convolutions, and do not directly use the above implementation.^[30]

Convolution Layers

Standard convolution

By standard convolution we refer to the operation represented by the Conv1D/2D layer in Keras (Conv1d/2d in PyTorch). Depending on the `io_type` option used, there are two classes of implementations in hls4ml.^[30]

`io_parallel`

In Vivado/Vitis backends, parallel convolution relies on the `im2col` transformation of the input, which turns convolution into a matrix-multiplication task. This task is then implemented as a sequence of matrix-vector multiplications using the routine mentioned above. The Latency and Resource strategies refer to the function used for matrix-vector multiplication routine, with Resource allowing for a slightly larger models to be synthesized. Parallelism can be further controlled via the `ParallelizationFactor`. Catapult backend in turn uses a direct implementation of convolution via nested loops. The Quartus, oneAPI, and Catapult backends also implement a Winograd algorithm choosable by setting the implementation to Winograd or combination. Winograd implementation is available for only a handful of filter size configurations, and it is less concerned about bit accuracy and overflow. In certain conditions it can be faster.^[30]

`io_stream`

There are two main classes of `io_stream` implementations, `LineBuffer` and `Encoded`. `LineBuffer` is the default, and generally produces marginally better results, while Catapult and Vivado also implement `Encoded`, choosable with the `ConvImplementation` configuration option. In all cases, the data is processed serially, a pixel every time, with a pixel containing an array of all the channel values for the pixel.^[30]

Depthwise convolution

Depthwise implementation substitutes the matrix-vector multiplication in the kernel to the elementwise multiplication. The only implementation available is based on Latency strategy, used by both `io_parallel` and `io_stream`.^[30]

Pointwise convolution

Pointwise convolutions are a special case of convolution where the filter size is 1 for 1D or 1x1 for 2D.^[30]

For the Vivado/Vitis backends, there is a dedicated `io_parallel`/Latency strategy implementation of 1D pointwise convolutional layers. The reuse factor (RF) is used to split the layer execution and reuse the existing module RF times. The RF also limits the number of multipliers in each module. The initiation interval scales as the RF. One limitation is that it assumes `in_width` is divisible by the RF.^[30]

Activations

Most activations without extra parameters are represented with the Activation layer, and those with single parameters (leaky ReLU, thresholded ReLU, ELU) as ParametrizedActivation. PReLU has its own class because it has a parameter matrix (stored as a weight). The hard (piecewise linear) sigmoid and tanh functions are implemented in a HardActivation layer, and Softmax has its own layer class.^[30]

Backends have four softmax implementations that the user can choose from by setting the implementation parameter:^[30]

- **latency**: Good latency, but somewhat high resource usage. It does not work well if there are many output classes.
- **stable**: Slower but with better accuracy, useful in scenarios where higher accuracy is needed.
- **legacy**: An older implementation with poor accuracy, but good performance. Usually the latency implementation is preferred.
- **argmax**: If you don't care about normalized outputs and only care about which one has the highest value, using argmax saves a lot of resources. This sets the highest value to 1, the others to 0.^[30]

ModelGraph Class

This object can be used to perform common simulation and firmware-generation tasks. Below is a list of key methods that are directly accessible to users:^[30]

- compile
- predict
- write
- build
- trace

write method

Write the ModelGraph to the output directory specified in the config:

```
hls_model.write()
```

compile method

Compiles the written C++/HLS code and links it into the Python runtime. Compiled model can be used to evaluate performance (accuracy) through predict() method.^[30]

```
hls_model.compile()
```

predict method

In a manner analogous to Keras's predict API, predictions can be obtained simply by providing an input in the form of a NumPy array:^[30]

```
# Suppose that you already have input array X
# Note that you have to do hls_model.compile() before using predict
y = hls_model.predict(X)
```

build method

This method “builds” the generated HLS project. The parameters of build are backend-specific and usually include simulation and synthesis. Refer to each backend for a complete list of supported parameters to build().^[30]

```
report = hls_model.build()
```

```
#You can also read the report of the build
hls4ml.report.read_vivado_report('hls4ml_prj')
```

The returned report object will contain the result of build step, which may include C-simulation results, HLS synthesis estimates, co-simulation latency etc, depending on the backend used.^[30]

trace method

The trace method is an advanced version of the predict method. It’s used to trace individual outputs from each layer of the hls_model. This is useful for debugging and setting the appropriate configuration.^[30]

Return: A dictionary where the keys are the names of the layers, and its values are the layers’s outputs.^[30]

```
predict_outputs, trace_outputs = hls_model.trace(X)

#We also support a similar function for keras
keras_trace = hls4ml.model.profiling.get_ymodel_keras(keras_model,
X)
```

3.4 Python API Configuration

With the use of HLS4ML, quickly a configuration dictionary can be generated from a keras model.^[30]

```
import hls4ml
config = hls4ml.utils.config_from_keras_model(model,
granularity='model')
```

This python dictionary can be edited as needed. More advanced configuration can be generated by, for example for ONNX models:

```
import hls4ml
config = hls4ml.utils.config_from_onnx_model(
    model,
    granularity='name',
    default_precision='fixed<16,6>',
    backend='Vitis')
```

for Keras models:

```
import hls4ml
```

```
config = hls4ml.utils.config_from_keras_model(
    model,
    granularity='name',
    default_precision='fixed<16,6>',
    backend='oneAPI')
```

or for PyTorch models:

```
import hls4ml
config = hls4ml.utils.config_from_pytorch_model(
    model,
    granularity='name',
    default_precision='fixed<16,6>',
    backend='Catapult')
```

The name granularity includes per-layer configuration based on the model. A 'name' granularity is generally recommended because it allows for more turning, and also because it allows for automatic setting of precisions. The layer-level precisions with the 'name' granularity default to 'auto', which means that hls4ml will try to set it automatically. Note that layer-level settings take precedence over model-level settings. A 'name' granularity is required for QKeras and QONNX model parsing. Passing the backend to these functions is recommended because some configuration options depend on the backend.^[30]

One can override specific values before using the configuration:

```
config['LayerName']['fc1']['ReuseFactor'] = 2
```

Or you can change the precision of the weight:

```
config['LayerName']['fc1']['Precision']['weight'] = 'fixed<8,4>'
```

Finally, one then uses the configuration to create an hls model:

```
hls_model = hls4ml.converters.convert_from_keras_model(
    model,
    hls_config=config,
    output_dir="my_project_dir",
    io_type='io_stream',
    backend='Vitis'
)
```

Chapter 4: Implementation of Cardiac Arrhythmia Detection Models

Chapter 4 presents the implementation details of the neural network models developed for cardiac arrhythmia detection. It begins by describing the selection of the codebase used for the project, which was the "arrhythmia-detection" repository by Cristiano Aguiar.^[31] The chapter discusses modifications made to the original code, such as changes in data acquisition, model definitions, and the training and evaluation process. Next, the implementation of Convolutional Neural Networks (CNN) is covered, with a focus on different model architectures built and tested using TensorFlow and Keras. The chapter also explains how HLS4ML is employed to convert the trained models into hardware description language (HDL) for efficient FPGA deployment. Details of the quantization techniques applied to the models, including various bit precision experiments, are discussed, followed by the configuration and synthesis process using Vivado to ensure the models are optimized for low-power, resource-efficient hardware execution.

4.1 Codebase Selection

For the implementation of the Neural Networks models for cardiac arrhythmia detection, this work leveraged the open-source repository arrhythmia-detection by Cristiano Aguiar (Csaquiar)^[31], available at <https://github.com/csaguiar/arrhythmia-detection/tree/master>. This repository was selected due to its comprehensive implementation of ECG signal processing pipelines specifically designed for the MIT-BIH Arrhythmia Database, a freely available database on PhysioNet.^[32] The MIT-BIH Arrhythmia Database consists of 48 thirty-minute segments of two-channel ambulatory ECG recordings, collected from 47 individuals at the BIH Arrhythmia Laboratory between 1975 and 1979.^[33] This codebase was also selected because it provide a well-structured foundation that could be modified to incorporate hardware design principles essential for our low-cost implementation focus. Also this repository's clear separation of preprocessing, model definition, evaluation components, and its use of standard deep learning frameworks (TensorFlow/Keras), makes it particularly suitable for our hardware optimization pipeline.

4.1.1 Codebase Analysis

Cristiano Aguiar in his project uses Python programming language within Jupyter Notebook environments, enabling interactive development and efficient management of code and results. The codebase provides a modular and well-documented implementation of ECG signal preprocessing, model architectures, and evaluations, tailored for the MIT-BIH Arrhythmia Database.

- **Preprocessing pipeline:** The notebook implements detailed processing of the MIT-BIH ECG signals, including data loading, segmentation of the signals into normal and abnormal beats, normalization, and formatting the data into a form suitable for input to neural networks. It also handles the splitting of the dataset into training and validation subsets (75% for training and 25% for validation).

- **Model definition:** Two different models are defined in the notebook using TensorFlow/Keras. The first one is a CNN only model and the second is a CNN with LSTM model. Also for both models the Adam optimizer was used.
- **Training and evaluation scripts:** The notebook performs model training using the fit method from Keras. The epochs were set to 15 and batch size was set to 128. Performance metrics like accuracy and loss are monitored during training. The evaluation is done with the evaluate method from Keras using the data for validation.

4.1.2 Core Modifications

Preprocessing pipeline

One of the main modifications made to the original codebase concerns the acquisition and handling of the MIT-BIH Arrhythmia Database. While the original notebook automatically downloads the database using a URL, in this implementation the dataset was downloaded manually beforehand. The code was then adapted to check whether the data has already been extracted locally, proceeding with extraction only if necessary.

Original code:

```
database_filename = "mit-bih-arrhythmia-database-1.0.0.zip"
database_path = Path(database_filename)

if not database_path.exists():
    url = f'https://storage.googleapis.com/mitdb-1.0.0.physionet.org/{database_filename}'
    wget.download(url)
    with zipfile.ZipFile(database_filename, 'r') as zip_ref:
        zip_ref.extractall(".")
```

Modified code:

```
database_filename = "mit-bih-arrhythmia-database-1.0.0.zip"
expected_file = Path("mit-bih-arrhythmia-database-1.0.0")

if not expected_file.exists():
    with zipfile.ZipFile(database_filename, 'r') as zip_ref:
        zip_ref.extractall(".")
    print("Extraction complete.")
else:
    print("Files already extracted.")
```

Model definition

The original notebook defined two models: a basic CNN and a hybrid CNN-LSTM. For this project, these were replaced by the four custom-designed CNN architectures described in Chapter 4.3.

Training and evaluation scripts

Significant changes were made to the training scripts to enable saving the trained model along with the training history of accuracy and loss. This allows the model to be trained once, saving considerable time in repeated executions. Specifically, training is controlled by a train boolean variable. When train=True, the model is trained, saved as a .keras file and the training history is saved to a .json file. When train=False, the pretrained model and training history are loaded directly, bypassing retraining. The evaluation cell didn't change.

Original code:

```
hist_cnn = cnn_model.fit(
    X_train,
    y_train,
    batch_size=128,
    epochs=15,
    validation_data=(X_val, y_val)
)
```

Modified code:

```
train = False
if train:
    hist_cnn = cnn_model.fit(
        X_train,
        y_train,
        batch_size=128,
        epochs=15,
        validation_data=(X_val, y_val)
    )
    cnn_model.save('NEW_model/cnn_model.keras')

    with open('NEW_model/hist_cnn.json', 'w') as f:
        json.dump(hist_cnn.history, f)

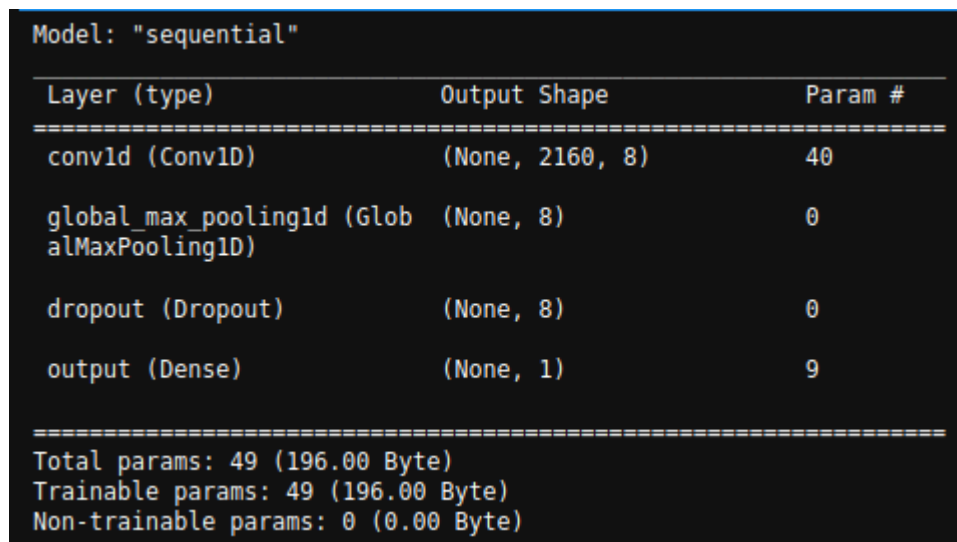
    with open('NEW_model/hist_cnn.json', 'r') as f:
        hist_cnn = json.load(f)
else:
    cnn_model = load_model('NEW_model/cnn_model.keras')

    with open('NEW_model/hist_cnn.json', 'r') as f:
        hist_cnn = json.load(f)
```

4.2 CNN Models and Frameworks

In developing the Convolutional Neural Network (CNN) models employed in this work, the TensorFlow 2.15.1 and Keras 2.15.0 was used. TensorFlow provides an extensive suite of tools for defining, training, and evaluating deep learning models in a flexible and efficient manner, while Keras offers a user-friendly interface that simplifies the construction of neural network architectures^{[34][35]}. Using these tools 3 CNN architectures (Model 1, Model 2 and Model 3) were developed and evaluated, with each model achieving higher accuracy than the previous. A fourth model (Model 4) was also tested which is the same model as the Model 3 but during the training the model is pruned to 50% sparsity. The models can be summarized as follows:

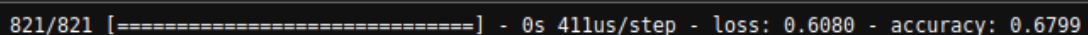
Model 1: A relatively simple CNN architecture serving as a baseline. It consists of one convolutional 1D layer, followed by GlobalMaxPooling1D, a Dropout and a Dense layer. The model applies a ReLU activation function in the standard 1D convolutional layer and a sigmoid activation in the dense layer. Optimization is performed using the Adam algorithm with a learning rate of 0.001. Following training and evaluation, Model 1 achieved an accuracy of 67,99%.

A screenshot of a terminal window showing the Keras model summary for a sequential model. The model has four layers: conv1d (Conv1D), global_max_pooling1d (GlobalMaxPooling1D), dropout (Dropout), and output (Dense). The summary table shows the output shape and number of parameters for each layer. Below the table, it shows the total, trainable, and non-trainable parameters and their sizes in bytes.

Layer (type)	Output Shape	Param #
conv1d (Conv1D)	(None, 2160, 8)	40
global_max_pooling1d (GlobalMaxPooling1D)	(None, 8)	0
dropout (Dropout)	(None, 8)	0
output (Dense)	(None, 1)	9

=====
Total params: 49 (196.00 Byte)
Trainable params: 49 (196.00 Byte)
Non-trainable params: 0 (0.00 Byte)

Figure 19: Model 1

A screenshot of a Keras training progress bar. It shows the current step as 821/821, followed by a series of equals signs. To the right, it displays the time per step as 0s 411us/step, the loss as 0.6080, and the accuracy as 0.6799.

821/821 [=====] - 0s 411us/step - loss: 0.6080 - accuracy: 0.6799

Figure 20: Model 1 accuracy after evaluation.

Model 2: The Model 2 is an extended version of Model 1, adding more complexity in order to achieve higher accuracy. This model consist of 3 convolutional 1D layers, and each one of these is followed by a BatchNormalization layer, an Activation layer using ReLu and a MaxPooling1D layer. Then there is a GlobalMaxPooling1D layer. After there are two Dense layers, each one of these is followed by BatchNormalization layer, an Activation layer using ReLu and a Dropout layer. Last there is another Dense layer followed by an Activation layer using sigmoid. For

optimizer uses the Adam algorithm with learning rate of 0,001. After training the Model 2 manage to achieve 78,61% accuracy.

```

Adding convolutional block 0 with N=8 filters
Adding convolutional block 1 with N=16 filters
Adding convolutional block 2 with N=32 filters
Adding dense block 0 with N=32 neurons
Adding dense block 1 with N=16 neurons
Model: "keras_cnn_ld_hls4ml"

```

Layer (type)	Output Shape	Param #
input_1 (InputLayer)	[(None, 2160, 1)]	0
conv_0 (Conv1D)	(None, 2160, 8)	24
bn_conv_0 (BatchNormalizat ion)	(None, 2160, 8)	32
conv_act_0 (Activation)	(None, 2160, 8)	0
pool_0 (MaxPooling1D)	(None, 1080, 8)	0
conv_1 (Conv1D)	(None, 1080, 16)	384
bn_conv_1 (BatchNormalizat ion)	(None, 1080, 16)	64
conv_act_1 (Activation)	(None, 1080, 16)	0
pool_1 (MaxPooling1D)	(None, 540, 16)	0
conv_2 (Conv1D)	(None, 540, 32)	1536
bn_conv_2 (BatchNormalizat ion)	(None, 540, 32)	128
conv_act_2 (Activation)	(None, 540, 32)	0
pool_2 (MaxPooling1D)	(None, 270, 32)	0
GlobalPool (GlobalMaxPooli ng1D)	(None, 32)	0
dense_0 (Dense)	(None, 32)	1024
bn_dense_0 (BatchNormaliza tion)	(None, 32)	128
dense_act_0 (Activation)	(None, 32)	0
dropout (Dropout)	(None, 32)	0
dense_1 (Dense)	(None, 16)	512
bn_dense_1 (BatchNormaliza tion)	(None, 16)	64
dense_act_1 (Activation)	(None, 16)	0
dropout_1 (Dropout)	(None, 16)	0
output_dense (Dense)	(None, 1)	17
output_sigmoid (Activation)	(None, 1)	0

```

=====
Total params: 3913 (15.29 KB)
Trainable params: 3705 (14.47 KB)
Non-trainable params: 208 (832.00 Byte)

```

Figure 21: Model 2

```
821/821 [=====] - 1s 1ms/step - loss: 0.9682 - accuracy: 0.7861
```

Figure 22: Model 2 accuracy after evaluation

Model 3: The Model 3 is a different approach according to a achieve even higher accuracy. This model has a conventional 1D layer, followed by a MaxPooling1D layer, a Flatten layer, a Dropout and a Dense layer. The conventional layer uses ReLu activation and the Dense layer uses sigmoid activation. For optimizer uses the Adam algorithm with learning rate of 0,001. After training the Model 3 manage to achieve 83,25% accuracy, the higher of the 4 models.

```
Model: "sequential"
```

Layer (type)	Output Shape	Param #
conv1d (Conv1D)	(None, 2160, 16)	80
max_pooling1d (MaxPooling1D)	(None, 540, 16)	0
flatten (Flatten)	(None, 8640)	0
dropout (Dropout)	(None, 8640)	0
output (Dense)	(None, 1)	8641

```

=====
Total params: 8721 (34.07 KB)
Trainable params: 8721 (34.07 KB)
Non-trainable params: 0 (0.00 Byte)

```

Figure 23: Model 3

```
821/821 [=====] - 6s 7ms/step - loss: 0.6223 - accuracy: 0.8325
```

Figure 24: Model 3 accuracy after evaluation.

Model 4: The Model 4 is the same as the model 3 but during the training it is pruned to 50% sparsity. The tensorflow_model_optimization 0.7.5 library was use to perform the pruning during the training^[36]. The pruning is performed every two epoches. After training the Model 4 manage to achieve 82,45% accuracy. As we can see it the accuracy loss is very small. This is the code used for the pruning during the training:

```
import tensorflow_model_optimization as tfmot
```

```

from tensorflow_model_optimization.sparsity import keras as sparsity
from tensorflow.keras.models import load_model
from tensorflow.keras.optimizers import Adam
from tensorflow_model_optimization.sparsity.keras import
UpdatePruningStep
import json

# Define the pruning parameters
NSTEPS = int(X_train.shape[0]) // 128

pruning_params = {
    'pruning_schedule': sparsity.PolynomialDecay(
        initial_sparsity=0.0,
        final_sparsity=0.5,
        begin_step=NSTEPS * 2,
        end_step=NSTEPS * 15,
        frequency=NSTEPS
    )
}

# Function to apply pruning to the layers
def pruneFunction(layer):
    if isinstance(layer,
tfmot.sparsity.keras.prune_wrapper.PruneLowMagnitude): # skip
already pruned
        return layer
    if isinstance(layer, tfmot.sparsity.keras.PrunableLayer):
        if isinstance(layer, Dense) and layer.name == 'output':
            return layer
        return tfmot.sparsity.keras.prune_low_magnitude(layer,
**pruning_params)
    if isinstance(layer, Conv1D):
        return tfmot.sparsity.keras.prune_low_magnitude(layer,
**pruning_params)
    return layer

# Apply pruning to the model
pruned_model = tfmot.sparsity.keras.prune_low_magnitude(
    cnn_model,
    **pruning_params
)

# Compile the pruned model
optimizer = Adam(learning_rate=0.001)
pruned_model.compile(
    optimizer=optimizer,
    loss="binary_crossentropy",
    metrics=["accuracy"]
)

pruned_model.summary()

pruning_callbacks = [UpdatePruningStep()]

train = True
if train:
    hist_cnn = pruned_model.fit(
        X_train,
        y_train,

```

```

        batch_size=128,
        epochs=15,
        validation_data=(X_val, y_val),
        callbacks=pruning_callbacks # Add the pruning callback
    )
    pruned_model.save('Model4_prune/pruned_cnn_model.keras')

    with open('Model4_prune/hist_cnn.json', 'w') as f:
        json.dump(hist_cnn.history, f)

    with open('Model4_prune/hist_cnn.json', 'r') as f:
        hist_cnn = json.load(f)
else:

    from qkeras.utils import _add_supported_quantized_objects
    from tensorflow_model_optimization.python.core.sparsity.keras
import pruning_wrapper

    co = {}
    _add_supported_quantized_objects(co)
    co['PruneLowMagnitude'] = pruning_wrapper.PruneLowMagnitude

    # Load pre-trained pruned model and history
    pruned_model = load_model('Model4_prune/pruned_cnn_model.keras',
custom_objects=co)

    with open('Model4_prune/hist_cnn.json', 'r') as f:
        hist_cnn = json.load(f)

```

```
Model: "sequential"
```

Layer (type)	Output Shape	Param #
prune_low_magnitude_conv1d (PruneLowMagnitude)	(None, 2160, 16)	146
prune_low_magnitude_max_pooling1d (PruneLowMagnitude)	(None, 540, 16)	1
prune_low_magnitude_flatten (PruneLowMagnitude)	(None, 8640)	1
prune_low_magnitude_dropout (PruneLowMagnitude)	(None, 8640)	1
prune_low_magnitude_output (PruneLowMagnitude)	(None, 1)	17283

```

=====
Total params: 17432 (68.11 KB)
Trainable params: 8721 (34.07 KB)
Non-trainable params: 8711 (34.05 KB)
=====

```

Figure 25: Model 4

```
821/821 [=====] - 2s 2ms/step - loss: 0.6177 - accuracy: 0.8245
```

Figure 26: Model 4 accuracy after training and pruning.

4.3 HLS4ML Implementation

The implementation of Neural Networks models on hardware level requires a specialized translation from High-Level Synthesis software (C or C++) models into Hardware Description Languages (Verilog or VHDL) optimized for efficient execution. In this context, the HLS4ML framework offers a powerful solution by automating the conversion of trained machine learning models into High-Level Synthesis (HLS) code. HLS4ML significantly reduces the time and complexity traditionally associated with hardware design, enabling rapid prototyping of low-latency, resource-efficient implementations. By leveraging fixed-point quantization and configurable hardware parameters, HLS4ML facilitates the balance between accuracy, resource utilization, and inference latency, making it an ideal tool for deploying deep learning models in embedded and real-time environments such as cardiac arrhythmia detection. In this project the 1.1.0 version of the HLS4ML is used. After creating the HLS code HLS4ML can export an IP Block with the necessary Verilog, ready to be imported in different software for synthesis and analysis of hardware description language designs like Vivado^[30].

4.3.1 Configuration and Model Conversion

Using the HLS4ML we can easily create a configuration dictionary. In this configuration we will use the `granularity='name'` and `backend='Vitis'`. With granularity 'name' we have access to every layer of the Neural Network and change settings like Precision, ReuseFactor and Strategy. More information about the settings used for every model will be in the Chapter 4.3.2. With backend 'Vitis' the hls4ml will use the Vitis HLS tool in order to synthesizing a C/C++ code into RTL. Specifically the Vitis HLS version used in this project is the 2023.2^[37]. The configuration is created with this code for all 4 models:

```
config = hls4ml.utils.config_from_keras_model(cnn_model,
granularity='name', backend='Vitis')
```

After creating the configuration dictionary, a hls model is created using this function:

```
hls_model = hls4ml.converters.convert_from_keras_model(cnn_model,
hls_config=config, backend='Vitis', output_dir='hls4ml/model_2',
part='xcu250-figd2104-2L-e', io_type='io_stream')
```

The variable `cnn_model` corresponds to the Keras neural network model, while `hls_config` contains the generated configuration parameters. Setting `backend='Vitis'` enables the use of Vitis HLS for RTL conversion. The `output_dir` specifies the directory in which the translated HLS model is stored, and `part` defines the FPGA device targeted in the project. The `io_type` parameter determines how data is exchanged with the network and between its layers. In this work, the `io_stream` option is applied, which is suitable for large CNNs^[30]. Under this mode, data is transmitted one pixel at a time, with each pixel represented as an array of channels that are sent in parallel.

After creating the hls model we can plot it and see all the modifications made, using this function:

```
hls4ml.utils.plot_model(hls_model, show_shapes=True,
show_precision=True, to_file=None)
```

The `hls_model` is the name of the hls model which is plotted, the `show_shapes=True` displays shape information, the `show_precision=True` displays precision of layer's variables and the `to_file=None` means that the diagram is not saved as file and is displayed in the Jupyter Notebook.

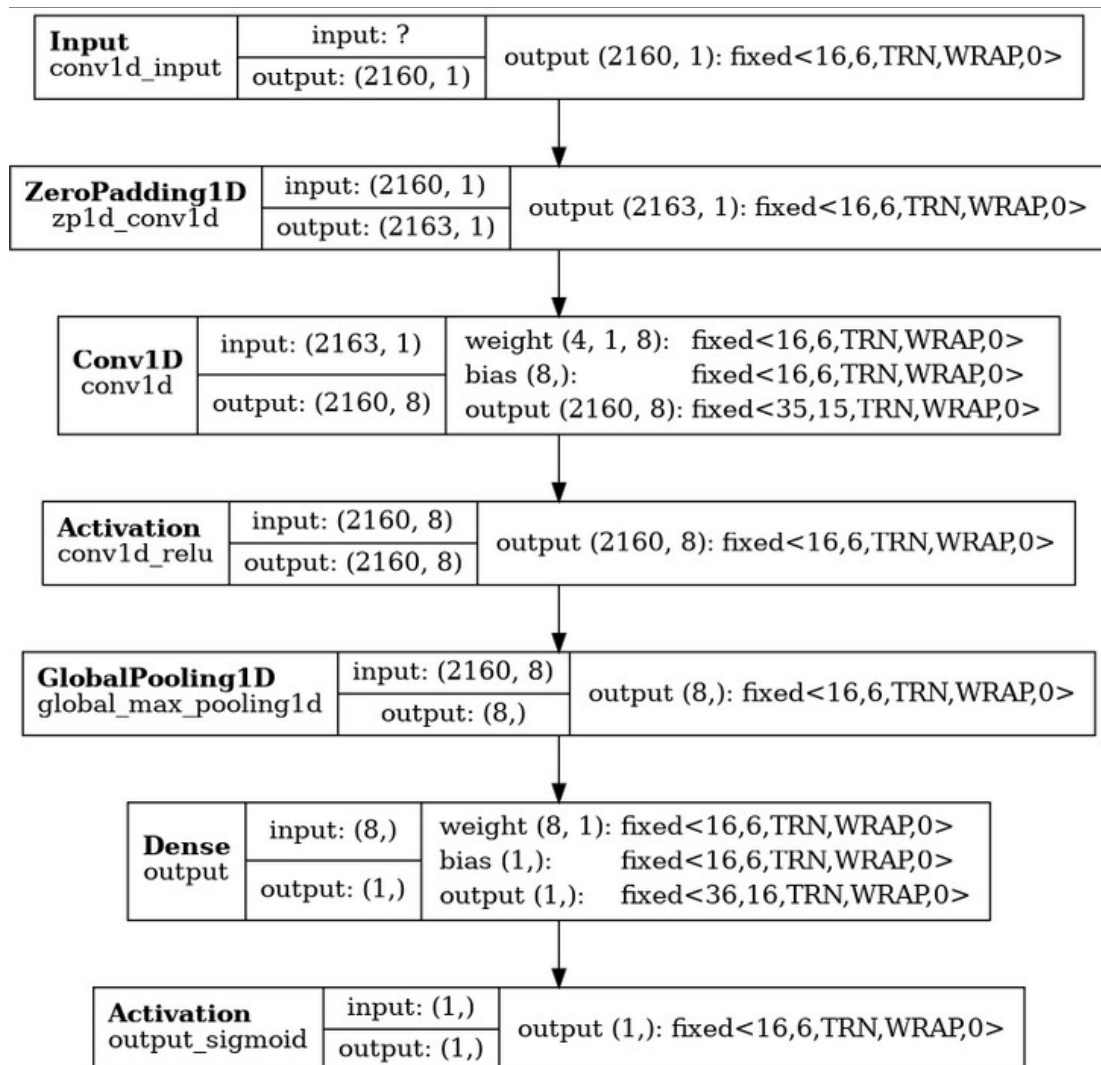


Figure 27: Diagram of Model 1 after conversion to HLS model.

The next part is to compile the hls model using this function:

```
hls_model.compile()
```

Then the compiled model can be used to check it's accuracy. The accuracy can be seen with this code:

```
from sklearn.metrics import accuracy_score

X_val = np.ascontiguousarray(X_val)
y_hls = hls_model.predict(X_val)

y_hls_pred = (y_hls > 0.5).astype("int32")
acc_hls = accuracy_score(y_val, y_hls_pred)
```

```
print("HLS4ML Model Accuracy: {:.2f}%".format(acc_hls * 100))
```

The four Models after the conversion to hls model manage to hold their accuracy:

- **Model 1 accuracy:** 68,01%
- **Model 2 accuracy:** 79,43%
- **Model 3 accuracy:** 83,25%
- **Model 4 accuracy:** 82,45%

Model 3 manage to achieve the larger accuracy 83,25%. Comparing this accuracy with the accuracy of the Bayani's and Kargar's LDCNN model mentioned in chapter 2.2.1 for the MIT-BIH Arrhythmia Database, it is lower by 16,13% but the parameters for the Model 3 are 8.721 while for the LDCNN the parameters are more than 950.000.^[15] On the other hand, Model 3 achieved 1,25% accuracy more than Aguiar's CNN model, also mentioned in chapter 2.2.1, while having less parameters.^[31]

Then the hls model is ready to be synthesized, using the Vitis HLS. This is done with this function:

```
hls_model.build(csim=False, synth=True, vsynth=True, export=True)
```

The csim=False means that the C simulation is not performed and it is not needed because it is performed during the compile of the hls model. Setting synth=True initiates the HLS compilation, where the C/C++ model code is transformed into a RTL. Setting vsynth=True flag triggers the subsequent Vivado synthesis step, which further optimizes and synthesizes the RTL design using the Xilinx Vivado Design Suite, producing detailed reports on resource utilization, timing, and performance metrics. Finally, the export=True creates and save an IP Block ready to be used in software for synthesis and analysis.

Finally we can the the report generated from the .build function, using this function:

```
hls4ml.report.read_vivado_report('hls4ml/model_4/')
```

4.3.2 Quantization Implementation

The quantization performed to the Models of the project is separated in three form of quantization and is performed on different bits until the accuracy of the models starts to drops. More information about quantization can be found in the chapter 2.3.1. The first quantization is PTQ using HLS4ML and in the weights and bias of convolutional and dense layers of the Models. The second quantization is also PTQ using HLS4ML but to every parameter of the Models. The third and last quantization is QAT using QKeras also in every parameter of the Models, and was used to quantize the Model 3 and the Model 4. From now on, the three different quantization performed in the project will be called Quantization 1 (Q1), Quantization 2 (Q2) and Quantization 3 (Q3).

For the Model 3 and Model 4 the strategy of the configuration is set to 'Resources'. That is because Vivado's compiler has a fixed limit for trainable

parameters for each layer. The limit is 4096 parameter, beyond this threshold, fully unrolled (parallelized) compilation is no longer feasible. This is the code used for this reason:

```
config['Model']['Strategy'] = 'Resources'
```

The rest configuration setting are the same for all the Models except of precision which is tuned of the quantization needed to performed each time.

Quantization 1

For the Q1 the precision of the weights and bias for the convolutional and dense layers must be changed, based on the bits tested. This precision can be set with this code:

```
config['LayerName']['conv1d']['Precision']['weight'] =  
'ap_fixed<12,4>'
```

'LayerName', 'Precision' and 'weight' are the keys to sett the precision needed for a specific parameter of a layer. 'ap_fixed<12,4>' is the precision which will be sett. The first number is the bits used and the second number is the bits used for the integer. Different precisions was used until the accuracy of the Models starts to drop.

Model 1

For the Model 1, three different precision was tested <12,4>, <8,4> and <4,2>. The HLS Model manage to have near 0 accuracy drop for all the precision tested, around 67,92%. This is configuration for Model 1:

```
config['LayerName']['conv1d']['Precision']['weight'] =  
'ap_fixed<12,4>'  
config['LayerName']['conv1d']['Precision']['bias'] =  
'ap_fixed<12,4>'  
  
config['LayerName']['output']['Precision']['weight'] =  
'ap_fixed<12,4>'  
config['LayerName']['output']['Precision']['bias'] =  
'ap_fixed<12,4>'
```

The ap_fixed was set to the precision needed each time.

Model 2

For the Model 2, two different precision was tested <12,4> and <8,4>. The HLS Model's accuracy for <8,4> precision had dropped to 68,49%, a drop of 10,94%. This is the configuration for Model 2:

```

config['LayerName']['conv_0']['Precision']['weight'] =
'ap_fixed<12,4>'
config['LayerName']['conv_0']['Precision']['bias'] =
'ap_fixed<12,4>'

config['LayerName']['conv_1']['Precision']['weight'] =
'ap_fixed<12,4>'
config['LayerName']['conv_1']['Precision']['bias'] =
'ap_fixed<12,4>'

config['LayerName']['conv_2']['Precision']['weight'] =
'ap_fixed<12,4>'
config['LayerName']['conv_2']['Precision']['bias'] =
'ap_fixed<12,4>'

config['LayerName']['dense_0']['Precision']['weight'] =
'ap_fixed<12,4>'
config['LayerName']['dense_0']['Precision']['bias'] =
'ap_fixed<12,4>'

config['LayerName']['dense_1']['Precision']['weight'] =
'ap_fixed<12,4>'
config['LayerName']['dense_1']['Precision']['bias'] =
'ap_fixed<12,4>'

config['LayerName']['output_dense']['Precision']['weight'] =
'ap_fixed<12,4>'
config['LayerName']['output_dense']['Precision']['bias'] =
'ap_fixed<12,4>'

```

The ap_fixed was set to the precision needed each time.

Model 3

For the Model 3, two different precision was tested <12,4> and <8,4>. The HLS Model's accuracy for <8,4> precision had dropped to 69,05%, a drop of 14,2%. This is the configuration for Model 3:

```

config['LayerName']['conv1d']['Precision']['weight'] =
'ap_fixed<12,4>'
config['LayerName']['conv1d']['Precision']['bias'] =
'ap_fixed<12,4>'

config['LayerName']['output']['Precision']['weight'] =
'ap_fixed<12,4>'
config['LayerName']['output']['Precision']['bias'] =
'ap_fixed<12,4>'

```

The ap_fixed was set to the precision needed each time.

Model 4

For the Model 4, two different precision was tested $\langle 12,4 \rangle$ and $\langle 8,4 \rangle$. The HLS Model's accuracy for $\langle 8,4 \rangle$ precision had dropped to 78,9%, a drop of 3,55%. This is the configuration for Model 4:

```
config['LayerName']['conv1d']['Precision']['weight'] =  
'ap_fixed<12,4>'  
config['LayerName']['conv1d']['Precision']['bias'] =  
'ap_fixed<12,4>'  
  
config['LayerName']['output']['Precision']['weight'] =  
'ap_fixed<12,4>'  
config['LayerName']['output']['Precision']['bias'] =  
'ap_fixed<12,4>'
```

The `ap_fixed` was set to the precision needed each time.

Quantization 2

For the Q2 the precision of every parameter must be changed, based on the bits tested. This precision can be set with this code:

```
config['Model']['Precision'] = 'ap_fixed<12,4>'
```

'Model' and 'Precision' are the key needed to sett the precision for the parameter of the whole model. 'ap_fixed<12,4>' is the precision which will be sett.

Model 1

For the Model 1, three different precision was tested $\langle 12,4 \rangle$, $\langle 8,4 \rangle$ and $\langle 4,2 \rangle$. The HLS Model manage to have near 0 accuracy drop for all the precision tested, around 67,92%. This is configuration for Model 1:

```
config['Model']['Precision'] = 'ap_fixed<12,4>'
```

The `ap_fixed` was set to the precision needed each time.

Model 2

For the Model 2, two different precision was tested $\langle 12,4 \rangle$ and $\langle 8,4 \rangle$. The HLS Model's accuracy for $\langle 8,4 \rangle$ precision had dropped to 67,92%, a drop of 11,51%. This is the configuration for Model 2:

```
config['Model']['Precision'] = 'ap_fixed<12,4>'
```

The `ap_fixed` was set to the precision needed each time.

Model 3

For the Model 3, two different precision was tested <12,4> and <8,4>. The HLS Model's accuracy for <8,4> precision had drop to 69,23%, a dropped of 14,02%. This is the configuration for Model 3:

```
config['Model']['Precision'] = 'ap_fixed<12,4>'
```

The ap_fixed was set to the precision needed each time.

Model 4

For the Model 4, two different precision was tested <12,4> and <8,4>. The HLS Model's accuracy for <8,4> precision had dropped to 78,31%, a drop of 4,14%. This is the configuration for Model 4:

```
config['Model']['Precision'] = 'ap_fixed<12,4>'
```

The ap_fixed was set to the precision needed each time.

Quantization 3

For the Q3 the QKeras library was used, in order to perform QAT in the Model 3 and Model 4. For the QKeras library, version 0.9.0 was used^[38]. The code for the cells where Model 3 and Model 4 are defined, was modified in order to define the models with QKeras.

Model 3 TensorFlow/Keras code:

```
from keras.models import Sequential
from keras.layers import Conv1D, MaxPooling1D, Flatten, Dense,
Dropout
from keras.optimizers import Adam

sequence_size = X_train.shape[1]
n_features = 1

cnn_model = Sequential([
    Conv1D(
        filters=16,
        kernel_size=4,
        strides=1,
        input_shape=(sequence_size, n_features),
        padding="same",
        activation="relu"
    ),
    MaxPooling1D(pool_size=4), [ MaxPooling1D(pool_size=4),]
```

```

        Flatten(),
        Dropout(0.5),
        Dense(1, activation="sigmoid", name="output")
    ])

optimizer = Adam(learning_rate=0.001)

cnn_model.compile(
    optimizer=optimizer,
    loss="binary_crossentropy",
    metrics=["accuracy"]
)

cnn_model.summary()

```

Model 3 QKeras code:

```

from tensorflow.keras.layers import Input, Flatten, MaxPooling1D,
Dropout
from tensorflow.keras.models import Model
from qkeras import QConv1D, QActivation, QDense
from tensorflow.keras.optimizers import Adam

sequence_size = X_train.shape[1]
n_features = 1

inputs = Input(shape=(sequence_size, n_features))

# Quantized Conv1D
x = QConv1D(
    filters=16,
    kernel_size=4,
    strides=1,
    padding="same",
    kernel_quantizer="quantized_bits(12,0,alpha=1)",
    bias_quantizer="quantized_bits(12,0,alpha=1)",
    name="qconv1d"
)(inputs)
x = QActivation('quantized_relu(12,0)', name="qact_conv")(x)
x = MaxPooling1D(pool_size=4, name="maxpool")(x)
x = Flatten(name="flatten")(x)
x = Dropout(0.5, name="dropout")(x)

# Quantized Dense
x = QDense(
    1,
    kernel_quantizer="quantized_bits(12,0,alpha=1)",
    bias_quantizer="quantized_bits(12,0,alpha=1)",
    name="qdense"
)(x)
outputs = QActivation('quantized_sigmoid(12)', name="qact_output")
(x)

cnn_model = Model(inputs=inputs, outputs=outputs,
name="quantized_cnn")

optimizer = Adam(learning_rate=0.001)
cnn_model.compile(

```

```

optimizer=optimizer,
loss="binary_crossentropy",
metrics=["accuracy"]
)
cnn_model.summary()

from qkeras.autoqkeras.utils import print_qmodel_summary
print_qmodel_summary(cnn_model)

```

For the the Model 4 the same changes were made, since it uses the same model pruned with 50% sparsity. For the training cell the code stayed the same.

Model 3

For the Model 3, four different precision was tested 12bit, 8bit, 4bit and 2bit. This is the equivalent of using <12,1>, <8,1>, <4,1> and <2,1> precision in the HLS4ML configuration. The QKeras model's accuracy for 2bit precision had dropped to 78,63%, a drop of 4,62%. After the HLS conversion there was near 0 accuracy drop for each precision tested.

Model 4

For the Model 4, four different precision was tested 12bit, 8bit, 4bit and 2bit. This is the equivalent of using <12,1>, <8,1>, <4,1> and <2,1> precision in the HLS4ML configuration. The QKeras model's accuracy for 2bits precision had dropped to 75,38%, a drop of 7,07%. After the HLS conversion there was near 0 accuracy drop for each precision tested.

4.4 Vivado

Vivado is a software for synthesis and analysis of HDL designs. For this project Vivado 2023.2 version was used. Vivado allows the creation of a block design which makes possible designing with IP Integrators. Then with Vivado the block design can be turn into the necessary netlists by running synthesis and then the design can be implemented into the selected FPGA by running implementation. Implementation also allows for Timing, Power and Utilization Report^[38].

After creating a Vivado project, a Block Design can be created by pressing Create Block Design, in the Flow Navigator under the IP INTEGRATOR.

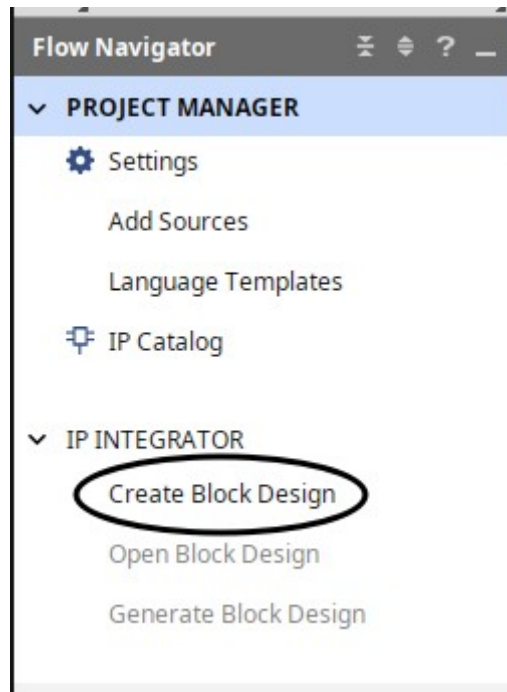


Figure 28: Create Block Design.

When the HLS4ML's function .build runs, a IP Block is generated with verilog code which represent the Neural Network behavior and it is saved in the selected folder. The IP Block can be imported in the Vivado project by selecting Tools>Settings and then in Project Settings IP>Repository. Then with the Add button the IP Block can be selected.

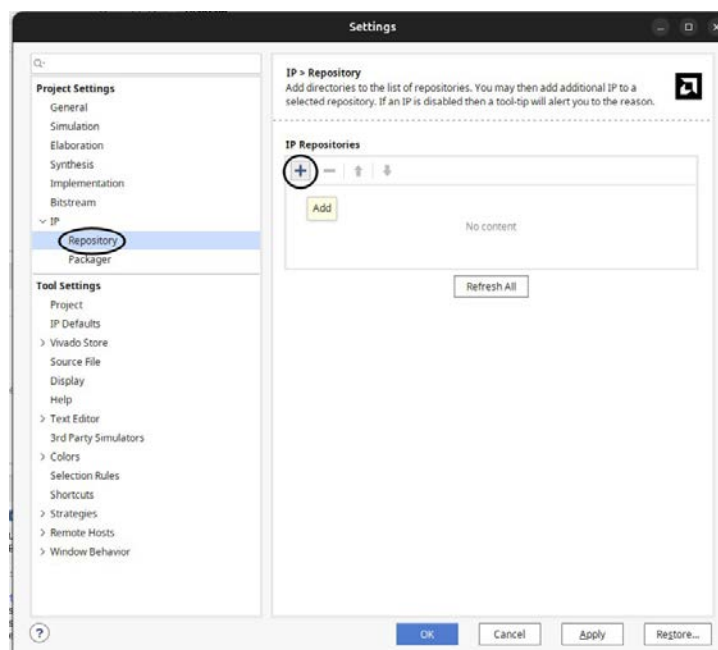


Figure 29: Import IP Block.

Then by pressing ctrl+I in the Diagram and by searching Myproject the IP Block is added to the Diagram of the Block Design.

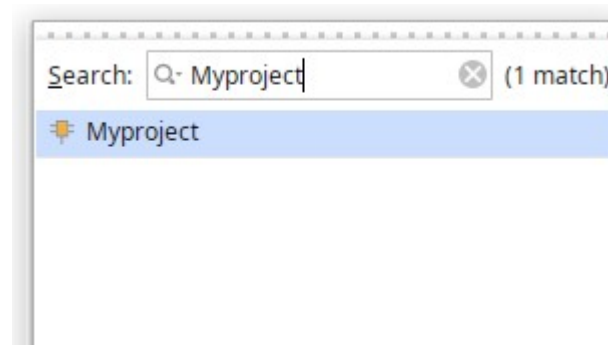


Figure 30: Add the IP Block to the Diagram.

Again by pressing ctrl+I four more IP Modules are added, Processor System Reset, Clocking Wizard and two Constants sett to high. After adding all the modules they are connected as shown in Figure 31.

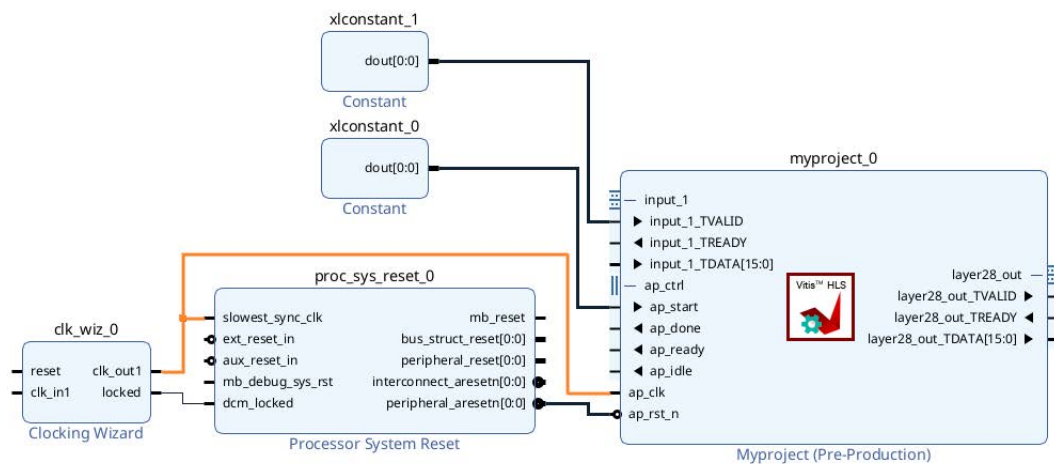
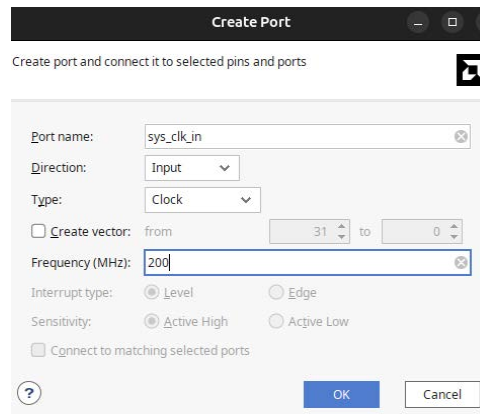


Figure 31: Modules connections

Then we add 4 Ports. One for clock, one for reset, one for the output of result data and one for result validation. A port is created by pressing right-click on diagram and then Create port... The used settings are shown on Figures 32-35 and the final connections are shown in the Figure 36.



Create Port

Create port and connect it to selected pins and ports

Port name: sys_clk_in

Direction: Input

Type: Clock

☐ Create vector: from 31 to 0

Frequency (MHz): 200

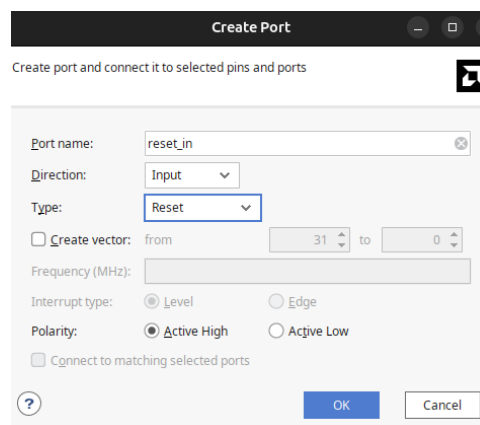
Interrupt type: ☒ Level ☐ Edge

Sensitivity: ☒ Active High ☐ Active Low

☐ Connect to matching selected ports

OK Cancel

Figure 32: Clock Settings.



Create Port

Create port and connect it to selected pins and ports

Port name: reset_in

Direction: Input

Type: Reset

☐ Create vector: from 31 to 0

Frequency (MHz):

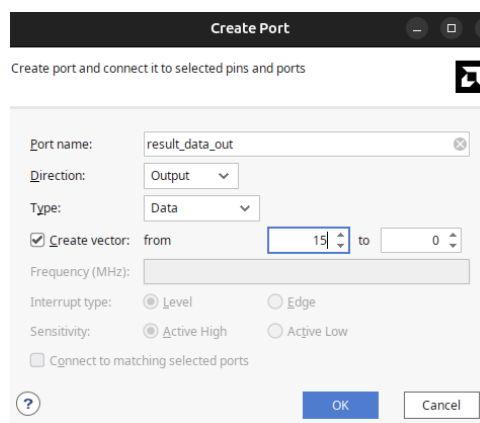
Interrupt type: ☒ Level ☐ Edge

Polarity: ☒ Active High ☐ Active Low

☐ Connect to matching selected ports

OK Cancel

Figure 33: Reset Settings.



Create Port

Create port and connect it to selected pins and ports

Port name: result_data_out

Direction: Output

Type: Data

☒ Create vector: from 15 to 0

Frequency (MHz):

Interrupt type: ☒ Level ☐ Edge

Sensitivity: ☒ Active High ☐ Active Low

☐ Connect to matching selected ports

OK Cancel

Figure 34: Output Data Settings.

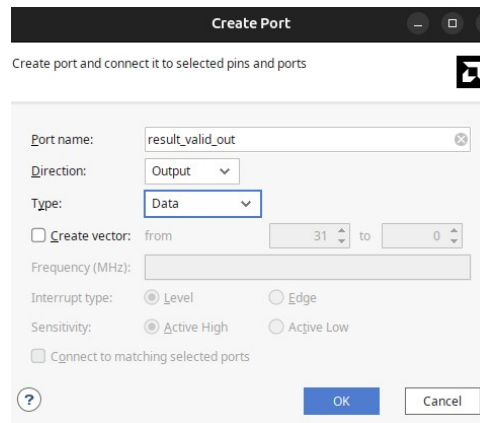


Figure 35: Result validation Settings.

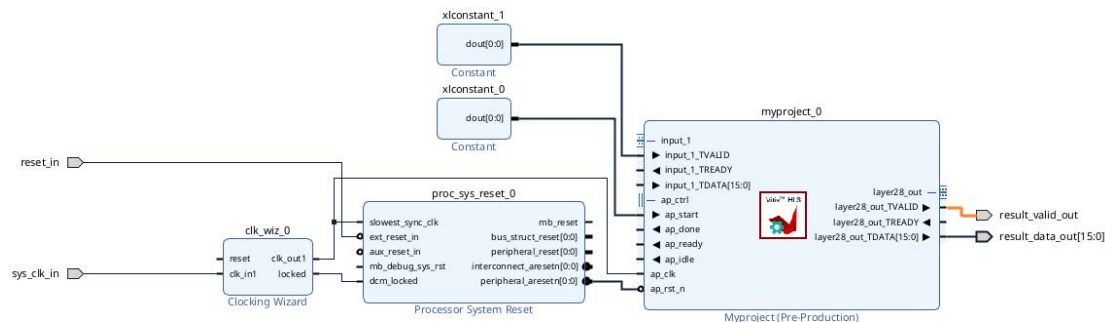


Figure 36: Final Block Design.

After the Block Design is ready F6 must be pressed to Validate the Design. If there is now error a Wrapper must be created by going on Sources window and the Design Sources>design_1. By right clicking and pressing Create HDL Wrapper... the Wrapper is created and the design is ready for synthesis. By pressing Run Synthesis on the Flow Navigator window under SYNTHESIS the necessary netlists are created. Then by pressing Run Implementation in the Flow Navigator window under IMPLEMENTATION the design is placed in the chosen FPGA and it is ready for Timing, Power and Utilization Reports.

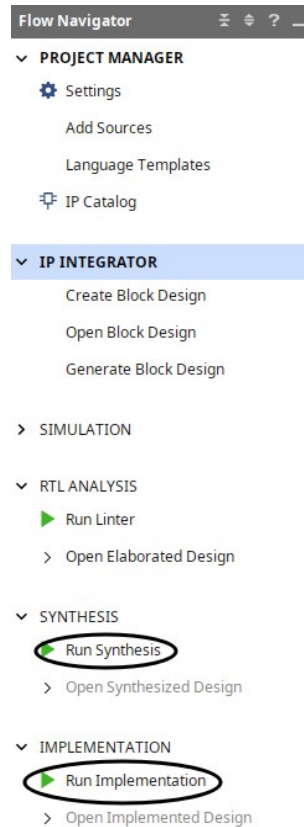


Figure 37: Run Synthesis and Run Implementation buttons.

When the Implementation is finished the Timing, Power and Utilization Report can be seen by pressing Report Timing Summary, Report Power and Report Utilization under the Open Implementation Design.

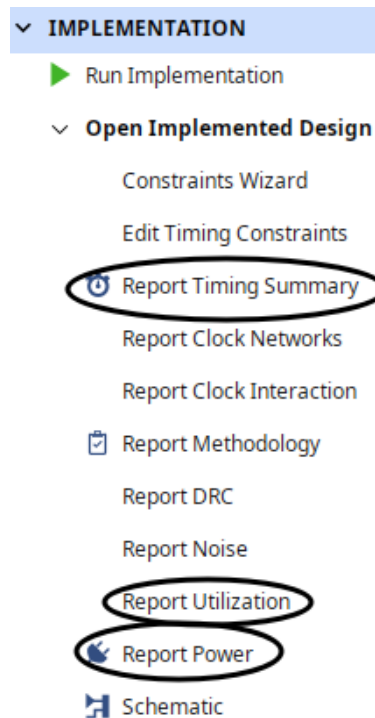


Figure 38: Report Timing Summary, Report Power and Report Utilization buttons.

This procedure is repeated for every precision tested in the three quantization types and for every model. In total this procedure was repeated 30 times.

4.5 Timing Improvement

After seeing the Timing, Power and Utilization Results from the previous steps, there are more steps taken for improving the Clock Period and seeing the results this change will have on the Timing, Power and Utilization.

The first step in order to change the Clock Period is to double click the Clocking Wizard module on the Block Diagram.

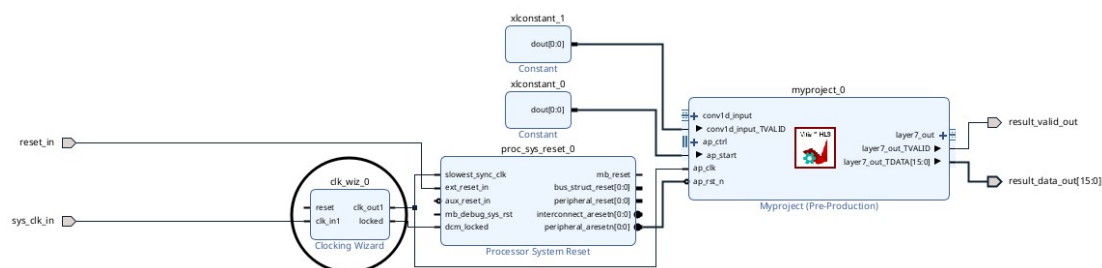


Figure 39: Clocking Wizard IP Module.

On the Re-customize IP window opened, the next step is to click the Output Clocks tab.

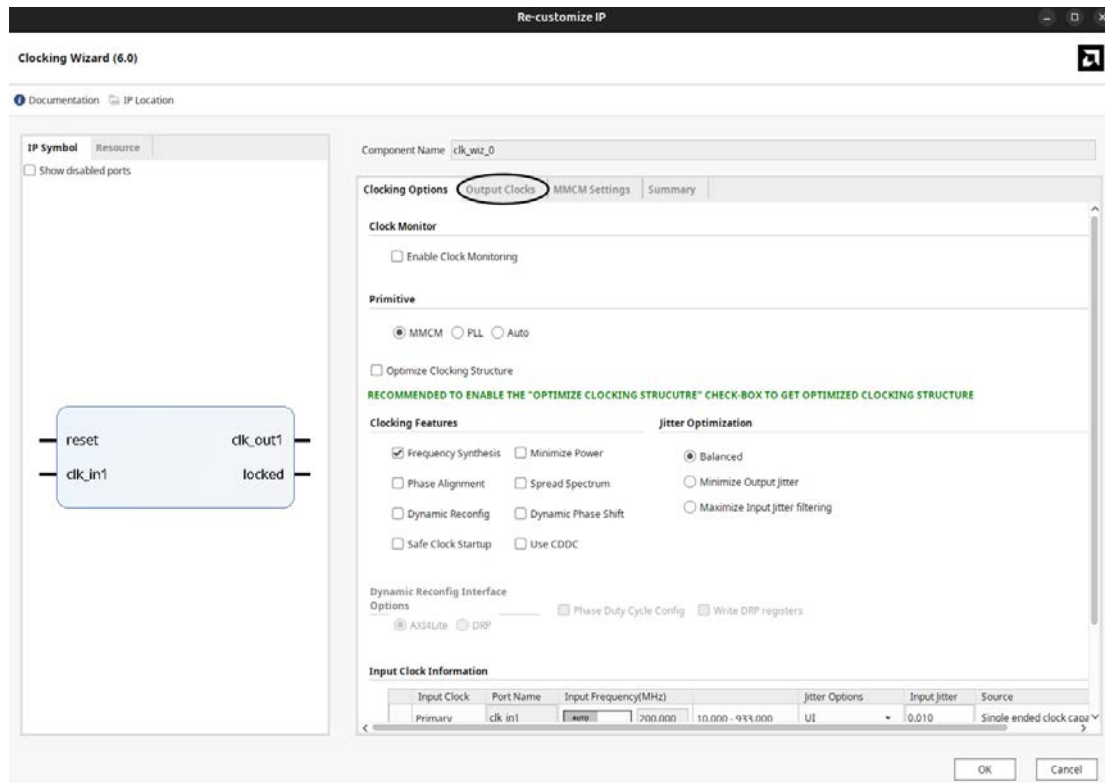


Figure 40: Output Clocks tab.

After going to the Output Clocks tab, the Clock Period can be changed by typing the new frequency needed in the Requested Output Freq section and clicking the OK button.

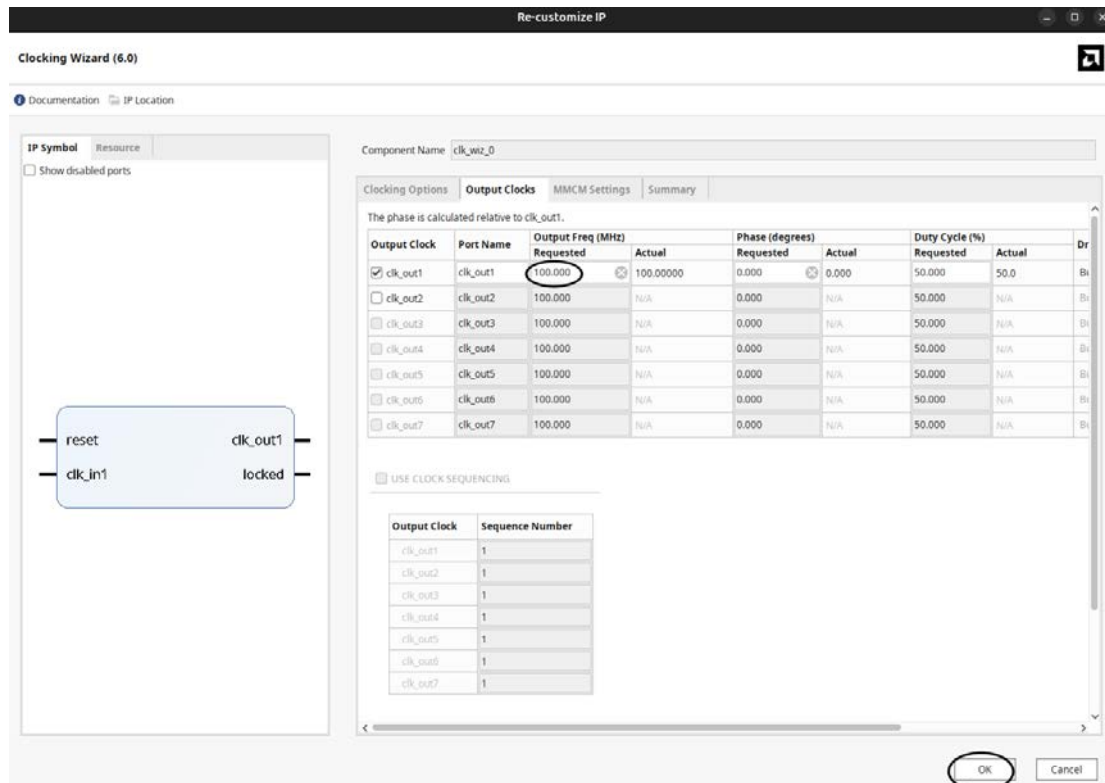


Figure 41: Requested Output Freq and OK button.

When the new Clock Period is set F6 must be pressed to Validate the Design. If there is now error a Wrapper must be created by going on Sources window and the Design Sources>design_1. By right clicking and pressing Create HDL Wrapper... the Wrapper is created and the design is ready for synthesis. By pressing Run Synthesis (Figure 37) on the Flow Navigator window under SYNTHESIS the necessary netlists are created. Then by pressing Run Implementation (Figure 37) in the Flow Navigator window under IMPLEMENTATION the design is placed in the chosen FPGA and it is ready for Timing, Power and Utilization Reports. The Reports can be seen by pressing Report Timing Summary, Report Utilization and Report Power (Figure 38).

Chapter 5: Results

This chapter presents the results obtained from the hardware implementation of the neural network models using the HLS4ML library and Vivado software^{[30][39]}. During quantization, different precisions were tested until the model tested starts dropping its accuracy. The accuracy achieved by the models is discussed in the Chapter 4.3.2. Furthermore, an analysis of the trade-offs between precision, hardware resources, and performance is provided, highlighting the practical implications for deploying low-latency, resource-efficient Neural Networks in embedded systems for cardiac arrhythmia detection.

5.1 Hardware Implementation Results

The hardware implementation of the Neural Networks Models was synthesized and evaluated using the Xilinx Vivado software, targeting the Alveo U250 (xcu250-fgd2104-2L-e) because it is one of the boards which HLS4ML supports. The synthesis reports provided data on the resource utilization, timing, and power consumption, which are critical metrics for assessing the feasibility and efficiency of deploying the models in real-time embedded applications. Specifically the metrics captured by the reports are these:

Utilization Report

- **LUT:** Lookup Table allow the creation of custom logic circuits, without the need for custom silicon.
- **FF:** Flip-Flop is a digital circuit with two stable states, capable of storing binary state information.
- **BRAM:** Block RAM is used to store large amount of data.
- **URAM:** Ultra RAM is a high density FPGA 288 Kb memory building block and coexist with BRAM on Xilinx's boards.
- **DSP:** Digital Signal Processor.
- **Carry8:** Carry8 is useful for building arithmetic functions.

Timing Report

- **WNS:** Worst Negative Slack is the time when the constraint isn't met. If it is negative there is setup violation.
- **WHS:** Worst Hold Slack is the smallest found in the design. If it is negative there is hold time violation.
- **WPWS:** Worst Pulse Width Slack is the smallest margin between the actual clock pulse width and the required minimum pulse width. If it is negative the pulse is too short somewhere and there is a violation.
- **Clock Period:** Clock Period is duration of one complete cycle of a clock signal.

Power Report

- **Total Power:** Total Power is the sum of all power consumed during the circuit operation.
- **Dynamic Power:** Dynamic Power is the portion of total power consumed by a digital circuit when its transistors switch states during operation.
- **Static Power:** Static Power is the power consumed by a digital circuit when it is powered on but not switching.
- **Clock Power:** Clock Power is the power consumed for the toggling of clock signals.
- **Signal Power:** Signal Power is the power consumed by all non-clock signals toggling.
- **Logic Power:** Logic Power is the power consumed by the logic elements.
- **BRAM Power:** BRAM Power is the power consumed by the BRAM.
- **URAM Power:** URAM Power is the power consumed by the URAM.
- **DSP power:** DSP Power is the power consumed by the DSPs.

The results of the Utilization Report are from NN's IP Block. The results of the Timing Report is for the entire design, including all the IP modules. The results for the Total Power, Dynamic Power and Static Power, are for the entire design including all the IP modules. The results for the Clock, Signal, Logic, BRAM, URAM and DSP Power are for NN's IP Block. The results are also separated in iso-delay and optimized-delays. The iso-delay is the a standard delay at 10ns and it is used to isolate the effect that quantization and pruning have on the models. The optimize-delays is the lowest delay achieved for each model and the effect it has on them.

Having established the key metrics and their definitions from the Vivado synthesis reports, the following sections present the detailed results obtained from the hardware implementation of the Neural Network models. These results highlight the resource consumption, timing performance, and power characteristics of the implemented designs.

5.1.1 Quantization 1

In Quantization 1, Post-Training Quantization (PTQ Chapter 2.3.1) is performed using the HLS4ML library to all models for different precision until the accuracy start to drop. The iso-delay was used for this quantization. In the Tables 1-3 are the results from the Timing, Power and Utilization Report.

Models	WNS	WHS	WPWS	Clock Period
Model 1	5,152	0,019	2,05	10
Model 1 <12,4>	4,635	0,017	2,05	10
Model 1 <8,4>	5,477	0,014	2,05	10
Model 1 <4,2>	6,221	0,019	2,05	10
Model 2	2,551	0,01	2,05	10
Model 2 <12,4>	3,565	0,01	2,05	10
Model 2 <8,4>	3,66	0,01	2,05	10

Model 3	0,321	0,01	2,05	10
Model 3 <12,4>	0,368	0,01	2,05	10
Model 3 <8,4>	0,557	0,01	2,05	10
Model 4	0,233	0,01	2,05	10
Model 4 <12,4>	1,624	0,01	2,05	10
Model 4 <8,4>	1,073	0,01	2,05	10

Table 1: Timing Report for Quantization 1 in nano-seconds.

Models	Total Power	Dynamic Power	Static Power	Clock Power	Signal Power	Logic Power	BRAM Power	URAM Power	DSP Power
Model 1	3,119	0,171	2,948	0,005	0,013	0,011	0,002	0,025	0,006
Model 1 <12,4>	3,108	0,16	2,948	0,004	0,011	0,01	0,002	0,024	0,005
Model 1 <8,4>	3,096	0,149	2,947	0,004	0,1	0,007	0,002	0,02	0
Model 1 <4,2>	3,087	0,139	2,947	0,003	0,006	0,005	0,002	0,02	0
Model 2	5,032	2,049	2,938	0,11	0,758	0,582	0,08	0	0,414
Model 2 <12,4>	4,569	1,595	2,975	0,095	0,554	0,511	0,085	0	0,243
Model 2 <8,4>	3,561	0,605	2,956	0,074	0,168	0,157	0,093	0	0,002
Model 3	5,65	2,654	2,996	0,304	0,857	0,734	0,005	0,049	0,599
Model 3 <12,4>	4,536	1,561	2,975	0,341	0,468	0,377	0,005	0,046	0,221
Model 3 <8,4>	3,563	0,607	2,956	0,245	0,147	0,063	0,005	0,041	0
Model 4	3,843	0,883	2,961	0,079	0,25	0,232	0,025	0	0,193
Model 4 <12,4>	3,628	0,671	2,957	0,091	0,175	0,169	0,024	0	0,109
Model 4 <8,4>	3,236	0,286	2,95	0,085	0,047	0,029	0,019	0	0

Table 2: Power Report for Quantization 1 in Watts.

Models	LUT	FF	BRAM	URAM	DSP	Carry8
Model 1	1200	1183	2,5	6	35	115
Model 1 <12,4>	1317	1190	2,5	5	3	181
Model 1 <8,4>	1473	1033	2,5	5	3	181
Model 1 <4,2>	1108	950	2,5	5	0	0
Model 2	57592	31867	70	0	2505	7205
Model 2 <12,4>	69695	31647	67	0	1560	8933
Model 2 <8,4>	61418	27835	63	0	11	8236
Model 3	124312	119517	5,5	12	3165	17538
Model 3 <12,4>	152674	173430	5,5	11	1154	22388
Model 3 <8,4>	88287	134260	5,5	10	3	10458
Model 4	28877	26008	28,5	0	1074	3734
Model 4 <12,4>	37373	40741	26,5	0	577	5117
Model 4 <8,4>	31044	44693	24,5	0	3	4072

Table 3: Utilization Report for Quantization 1.

5.1.2 Quantization 2

In Quantization 2, PTQ is performed using the HLS4ML library to all models for different precision until the accuracy start to drop. The iso-delay was used for this

quantization. In the Tables 4-6 are the results from the Timing, Power and Utilization Report.

Models	WNS	WHS	WPWS	Clock Period
Model 1	5,152	0,019	2,05	10
Model 1 <12,4>	4,922	0,003	2,05	10
Model 1 <8,4>	6,424	0,02	2,05	10
Model 1 <4,2>	7,085	0,018	2,05	10
Model 2	2,551	0,01	2,05	10
Model 2 <12,4>	3,523	0,01	2,05	10
Model 2 <8,4>	3,455	0,001	2,05	10
Model 3	0,321	0,01	2,05	10
Model 3 <12,4>	0,407	0,01	2,05	10
Model 3 <8,4>	0,708	0,1	2,05	10
Model 4	0,233	0,01	2,05	10
Model 4 <12,4>	2,022	0,01	2,05	10
Model 4 <8,4>	3,904	0,01	2,05	10

Table 4: Timing Report for Quantization 2 in nano-seconds.

Models	Total Power	Dynamic Power	Static Power	Clock Power	Signal Power	Logic Power	BRAM Power	URAM Power	DSP Power
Model 1	3,119	0,171	2,948	0,005	0,013	0,011	0,002	0,025	0,006
Model 1 <12,4>	3,107	0,16	2,948	0,006	0,012	0,009	0,001	0,021	0,005
Model 1 <8,4>	3,07	0,123	2,947	0,003	0,001	0,002	0,001	0,013	0
Model 1 <4,2>	3,065	0,119	2,947	0,003	0,002	0,001	0,005	0,007	0
Model 2	5,032	2,049	2,983	0,11	0,758	0,582	0,08	0	0,414
Model 2 <12,4>	4,483	1,51	2,973	0,087	0,499	0,472	0,063	0	0,285
Model 2 <8,4>	3,401	0,448	2,953	0,046	0,123	0,134	0,044	0	0
Model 3	5,65	2,654	2,996	0,304	0,857	0,734	0,005	0,049	0,599
Model 3 <12,4>	4,269	1,299	2,969	0,273	0,362	0,302	0,004	0,034	0,222
Model 3 <8,4>	3,305	0,353	2,951	0,138	0,051	0,031	0,002	0,03	0
Model 4	3,843	0,883	2,961	0,079	0,25	0,232	0,025	0	0,193
Model 4 <12,4>	3,556	0,601	2,956	0,075	0,146	0,135	0,016	0	0,126
Model 4 <8,4>	3,145	0,197	2,948	0,047	0,021	0,016	0,011	0	0

Table 5: Power Report in Quantization on Watts.

Models	LUT	FF	BRAM	URAM	DSP	Carry8
Model 1	1200	1183	2,5	6	35	115
Model 1 <12,4>	1254	1040	2	5	27	127
Model 1 <8,4>	1090	825	1,5	4	0	0
Model 1 <4,2>	571	549	5	2	0	0
Model 2	57592	31567	70	0	2505	7205
Model 2 <12,4>	56581	25913	54	0	1591	7235
Model 2 <8,4>	35228	15057	39	0	3	0
Model 3	124312	119517	5,5	12	3165	17538
Model 3 <12,4>	121481	132556	4,5	9	1151	17450
Model 3 <8,4>	47267	68413	3	7	0	0
Model 4	28877	26008	28,5	0	1074	3734
Model 4 <12,4>	29852	31567	21,5	0	576	4154
Model 4 <8,4>	16986	22990	15,5	0	0	2713

Table 6: Utilization Report for Quantization 2.

5.1.3 Quantization 3

In Quantization 3, Quantization-Aware Training (QAT Chapter 2.3.1) is performed using the QKeras library to all models for different precision until the accuracy start to drop. The iso-delay was used for this quantization. In the Tables 7-9 are the results from the Timing, Power and Utilization Report.

Models	WNS	WHS	WPWS	Clock Period
Model 3	0,321	0,01	2,05	10
Model 3 12bits	0,359	0,01	2,05	10
Model 3 8bits	0,565	0,01	2,05	10
Model 3 4bits	3,797	0,01	2,05	10
Model 3 2bits	4,603	0,006	2,05	10
Model 4	0,233	0,01	2,05	10
Model 4 12bits	0,742	0,005	2,05	10
Model 4 8bits	3,454	0,01	2,05	10
Model 4 4bits	4,875	0,01	2,05	10
Model 4 2bits	6,418	0,002	2,05	10

Table 7: Timing Report for Quantization 3 in nano-seconds.

Models	Total Power	Dynamic Power	Static Power	Clock Power	Signal Power	Logic Power	BRAM Power	URAM Power	DSP Power
Model 3	5,65	2,654	2,996	0,304	0,857	0,734	0,005	0,049	0,599
Model 3 12bits	6,037	3,034	3,003	0,272	0,995	0,936	0,004	0,038	0,678
Model 3 8bits	3,678	0,719	2,958	0,223	0,166	0,133	0,003	0,033	0,058
Model 3 4bits	3,208	0,259	2,95	0,071	0,032	0,024	0,002	0,027	0
Model 3 2bits	3,106	0,158	2,948	0,014	0,01	0,01	0,003	0,019	0
Model 4	3,843	0,883	2,961	0,079	0,25	0,232	0,025	0	0,193
Model 4 12bits	3,899	0,937	2,962	0,077	0,273	0,266	0,017	0	0,189
Model 4 8bits	3,277	0,326	2,95	0,065	0,057	0,055	0,013	0	0,031
Model 4 4bits	3,113	0,165	2,947	0,031	0,011	0,01	0,001	0	0
Model 4 2bits	3,076	0,13	2,947	0,011	0,005	0,005	0,007	0	0

Table 8: Power Report for Quantization 3 in Watts.

Models	LUT	FF	BRAM	URAM	DSP	Carry8
Model 3	124312	119517	5,5	12	3165	17538
Model 3 12bits	112214	85718	4	10	3875	15526
Model 3 8bits	79545	101637	3	8	311	9703
Model 3 4bits	23414	33507	2	7	0	1410
Model 3 2bits	4662	5517	3,5	5	0	241
Model 4	28877	26008	28,5	0	1074	3734
Model 4 12bits	31570	23580	23	0	1066	4058
Model 4 8bits	23097	28332	18,5	0	178	2773
Model 4 4bits	9219	12692	14	0	0	626
Model 4 2bits	3482	4071	10,5	0	0	171

Table 9: Utilization Report for Quantization 3.

5.2 Performance Analysis and Design Trade-offs

This section provides a detailed examination of the key performance metrics derived from the FPGA implementations of the Neural Network models under various quantization schemes and hardware configurations. The analysis focuses on the trade-offs between Utilization, Power consumption, Timing, and model Accuracy, which are critical factors in the deployment of efficient embedded machine learning systems.

5.2.1 Accuracy

In this project four Models was tested. Model 1 with 68,01% accuracy, Model 2 with 79,43% accuracy, Model 3 with 83,25% accuracy and a Pruned Model 4 with 82,45% accuracy. The Model 4 is the pruned version of Model 3 with 50% sparsity. There was an accuracy drop of only 0,8%.

The Model 1 was tested with Quantization 1 and Quantization 2. In both Quantizations the precision tested was <12,4>, <8,4> and <4,2>. The model manage to hold it's accuracy for the tested precisions. For <4,2> precision the model had 67,92% accuracy on both Quantizations as shown in the Figure 42. Smaller precision was not tested because the model was used as a baseline.

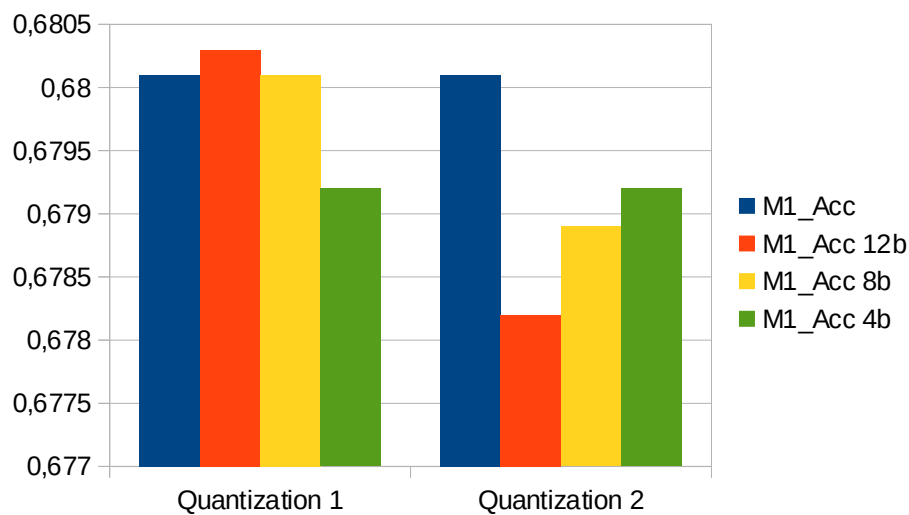


Figure 42: Model 1 accuracy.

The Model 2 was also tested with Quantization 1 and Quantization 2. Here two precision was tested, $\langle 12,4 \rangle$ and $\langle 8,4 \rangle$, for both Quantizations. The model dropped its accuracy on the $\langle 8,4 \rangle$ precision as shown in Figure 43. For the Q1 on this precision the accuracy was 68,49%, a drop of 10,94%. For the Q2 on this precision the accuracy was 67,92% a drop of 11,51%, 0,56% more than the Q1.

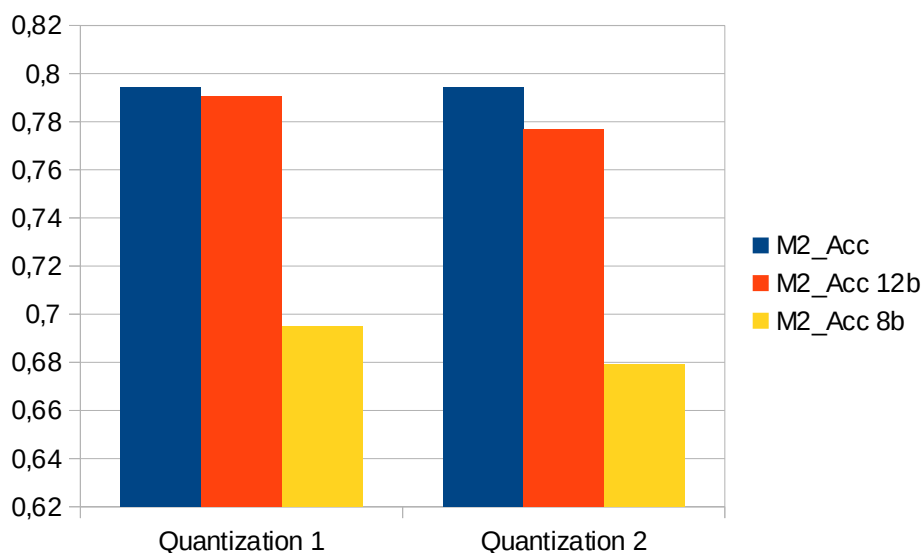


Figure 43: Model 2 accuracy.

The Model 3 was tested with Quantization 1, Quantization 2 and Quantization 3. For Q1 and Q2 the two precision was tested, $\langle 12,4 \rangle$ and $\langle 8,4 \rangle$, for both

Quantizations. The model dropped its accuracy on the $\langle 8,4 \rangle$ precision. For the Q1 on this precision the accuracy was 69,05%, a drop of 14,2%. For the Q2 on this precision the accuracy was 69,23%, a drop of 14,02%, 0,18% less than the Q1 as shown in Figure 44. For the Q3 four precision was tested $\langle 12,4 \rangle$, $\langle 8,4 \rangle$, $\langle 4,2 \rangle$ and $\langle 2,1 \rangle$. The model dropped its accuracy on the $\langle 2,1 \rangle$ precision. The accuracy on this precision was 78,63%, a drop of 4,62% as shown in Figure 44.

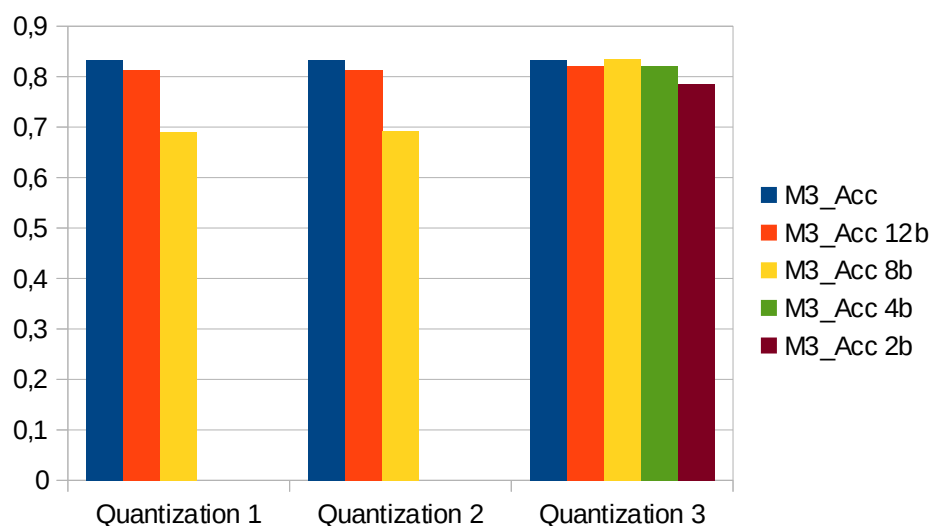


Figure 44: Model 3 accuracy.

The Model 4 was tested with Quantization 1, Quantization 2 and Quantization 3. For Q1 and Q2 the two precision was tested, $\langle 12,4 \rangle$ and $\langle 8,4 \rangle$, for both Quantizations. The model dropped its accuracy on the $\langle 8,4 \rangle$ precision. For the Q1 on this precision the accuracy was 78,9%, a drop of 3,55%. For the Q2 on this precision the accuracy was 78,31%, a drop of 4,14%, 0,59% more than the Q1 as shown in Figure 45. For the Q3 four precision was tested $\langle 12,4 \rangle$, $\langle 8,4 \rangle$, $\langle 4,2 \rangle$ and $\langle 2,1 \rangle$. The model dropped its accuracy on the $\langle 2,1 \rangle$ precision. The accuracy on this precision was 75,38%, a drop of 7,07% as shown in Figure 45.

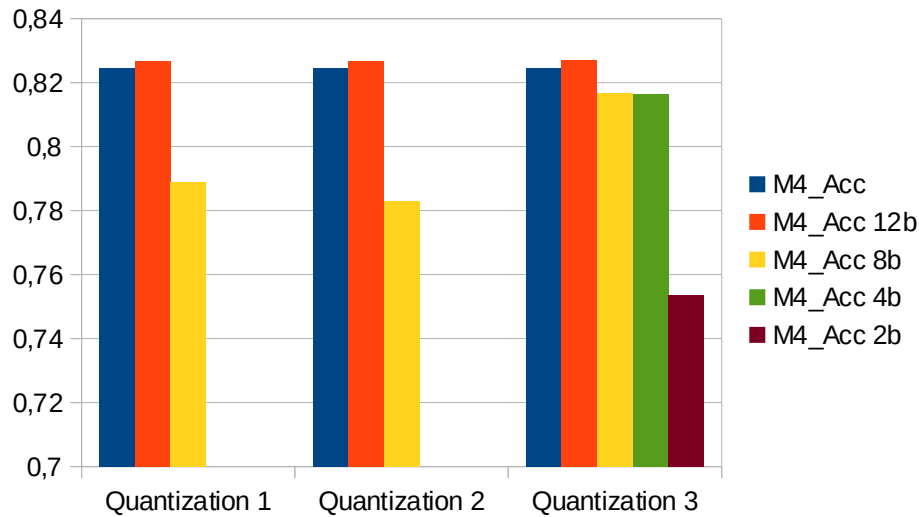


Figure 45: Model 4 accuracy.

As we can see for the PTQ techniques tested (Q1 and Q2) there was no significant difference when we are looking the accuracy. The models dropped their accuracy at the same precision for both Quantizations and none of the 2 techniques had better performance when we look the accuracy drop.

The PTQ techniques seem to perform better when pruning is used throughout the model's training phase. The Model 4 had only 3,55% drop for Q1 and 4,14% drop for Q2 while the Model 2 and Model 3 had 10,92% and 14,2% drop for Q1 and, 11,51% and 14,02% drop for Q2.

When it comes for the QAT technique, the models manage to be quantized in smaller precisions. Model 3 and Model 4 dropped their accuracy at 2 bit precision when performing QAT while the same models dropped their accuracy at 8 bit precision when performing PTQ. Also Model 3 had smaller accuracy drop at the 2bit QAT versus the 8bit PTQ.

From all the Quantizations tested the Quantization 3, (QAT with QKeras) seems to have the best results for the accuracy because much smaller precision was achieved.

5.2.2 Timing

From the Timing Report four values were examined, Clock Period, WNS, WHS and WPWS. The Clock Period is the iso-delay at 10ns and was used for all the quantizations and precision tested.

As show from the figures 42-44 WPWS was not affected for the different quantizations and precisions tested and stayed at 2,05ns. WNS value increases during quantization when the precision is lowered. The exceptions are the for Quantization 1 the Model 1 <12,4>, Model 4 <8,4> and for Quantization 2 the Model 1 <12,4>. The WHS during quantization sometimes was increasing and some time was decreasing.

This can be seen from the tables 1, 4 and 7. For instance, Model 1 has WHS 0,019ns and in Quantization 1 for <12,4> precision the WHS decreases to 0,017ns and in <4,2> precision goes back to 0,019ns.

The main purpose of the WNS, WHS and WPWS is to check for timing violation. If the values are positive there is no timing violation. During the tests their values were always positive.

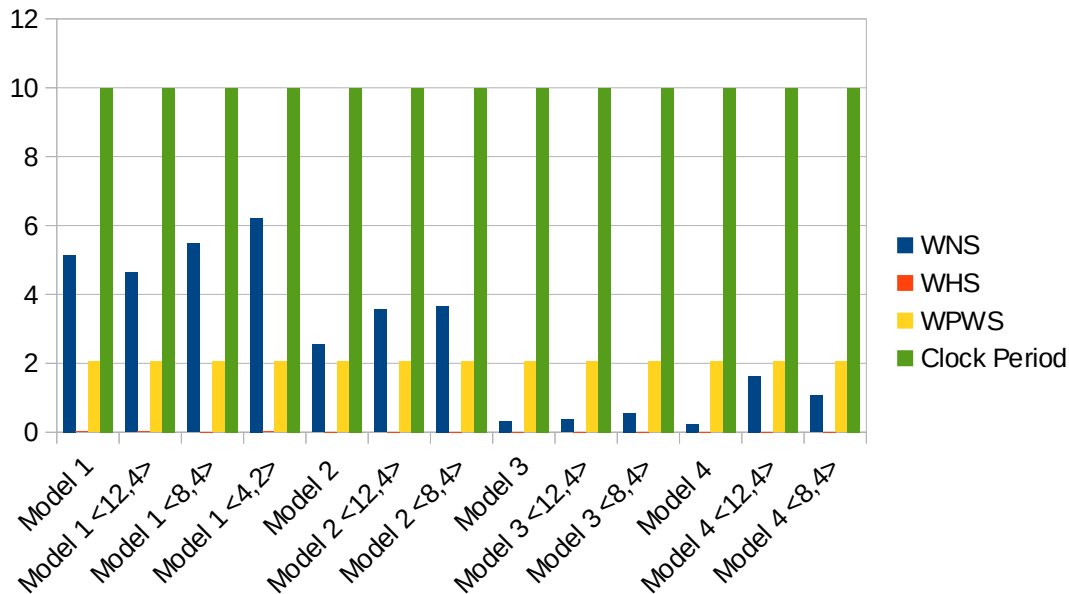


Figure 46: Timing trade offs for Quantization 1.

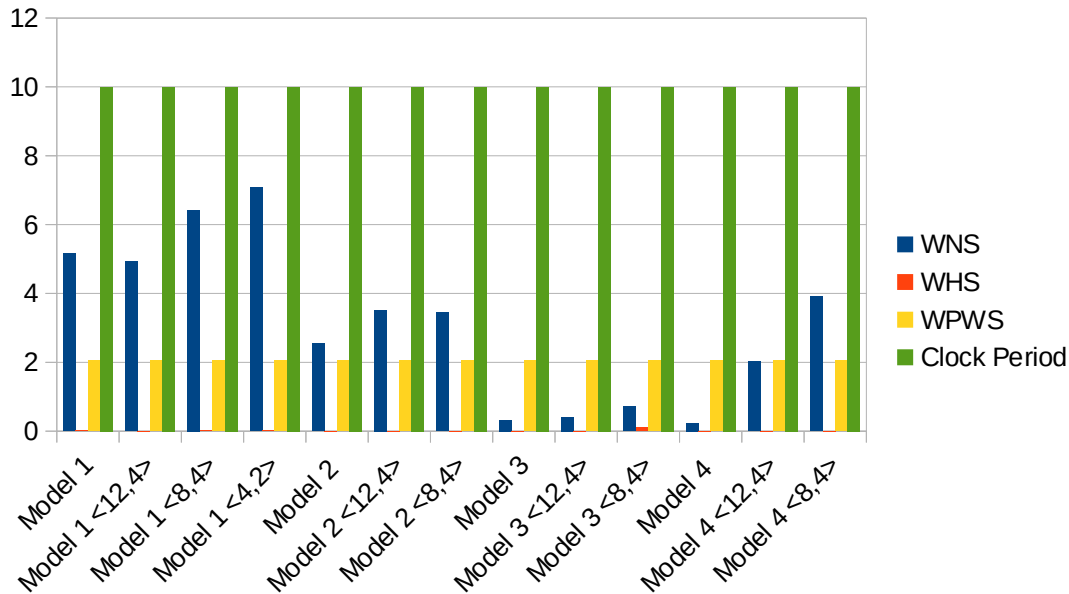


Figure 47: Time trade offs for Quantization 2.

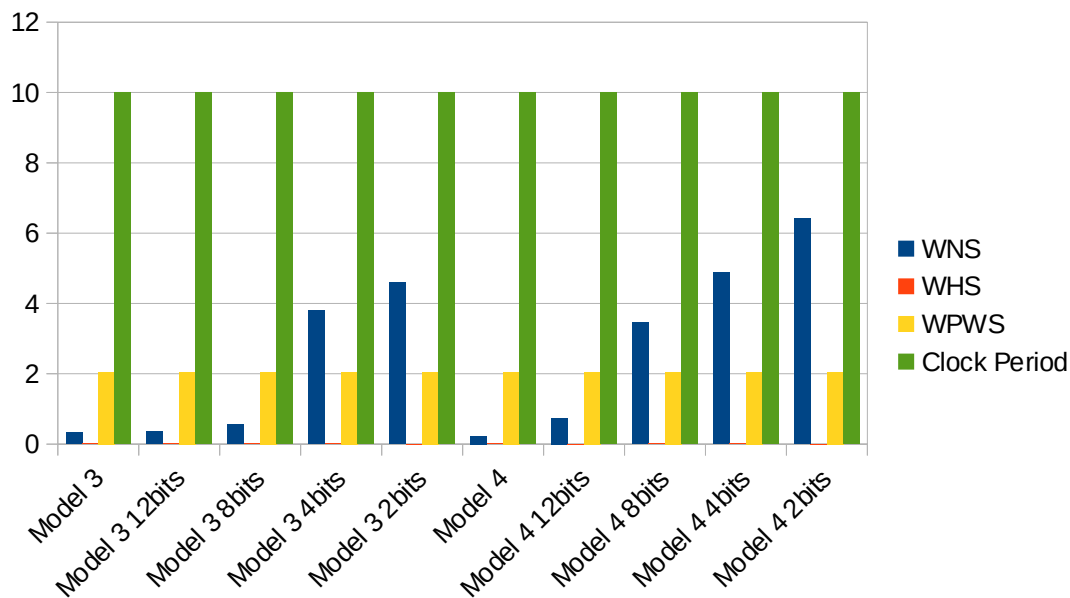


Figure 48: Time trade offs for Quantization 3.

5.2.3 Utilization

From the Utilization Report six values were examined, LUT, FF, BRAM, URAM, DSP and Carry8.

LUT: In most cases the number of LUTs used in the final precision was lowered. The two exceptions were in the Q1 for the Model 2 and Model 4. Originally they used 57.592 and 28.877 and after quantization on $\langle 8,4 \rangle$ precision they used 61.418 and 31.044. On average the Q1 had 5,62% drop, Q2 had 48,6% drop and Q3 had 92,1% drop.

FF: As with LUTs, the FFs used in the final precision was lowered. There are also two exceptions in Q1 for Model 3 and Model 4. Originally they used 119517 and 26.008 and after quantization on $\langle 8,4 \rangle$ precision they used 134.260 and 44.693 on average the Q1 had 12,96% increase, Q2 had 40,06% drop and Q3 had 89,87% drop.

BRAM: The BRAM in most cases decreases as the precisions also decreases. Because for the memory storage both BRAM and URAM are used the are cases where the extreme reduce of URAM usage, increase the usage of BRAM. This doesn't mean that the demand for memory is increased since the URAM can handle more amount of data than the BRAM. On average Q1 had 6,01% drop, Q2 had 8,84% drop and Q3 had 49,76% drop.

URAM: The URAM was not always used from the models. When it was used as the precision dropped the URAM always dropped. On average Q1 had 16,67% drop, Q2

had 54,17% drop and Q3 had 58,33% drop. The averages were calculated using only the models which used URAM before quantization.

DSP: The DSP usage dropped almost to 0 for all the quantizations. Specifically on average Q1 had 99,8% drop, Q2 had 99,98% drop and Q3 had 100% drop. As it seems when the model is quantized with precision 8bit or lower the need of DSPs can drop near 0.

Carry8: The Carry8 was also decreased in most cases except of Q1 for Model 2 and Model 4. In these cases the models originally had 7.205 and 3.734 and after quantization they had 8.236 and 4.072. On average the Q1 had 29,25% drop, Q2 had 81,84% drop and Q3 had 97,02% drop

It is clear from the results that the Quantization 3 which uses QAT technique with QKeras had significantly better performance than the other two techniques. It is also important to report that the number used for the averages was from the original Models and the final precision tested. When the average shows a drop doesn't mean that on the previous precision tested there was also a drop. For example the Q1 had 5,61% drop in LUTs usage but the original Model 3 uses 124.312 LUTs and when it is quantized with <12,4> precision it uses 152.674 LUTs.

Across all quantization flows, resource usage scales down with precision, with Quantization 3 delivering the most savings on average: LUTs -92,1%, FFs -89,87%, BRAM -49,76%, URAM -58,33%, Carry8 -97,02%, while DSPs are effectively eliminated -100%. Notably, Quantization1 shows a few exceptions where utilization increases at intermediate precisions e.g. LUTs rise for Model 2 and Model 4 at {8,4} (57,592→61,418 and 28,877→31,044) and FFs rise for Model 3 and Model 4 at {8,4} (119,517→134,260 and 26,008→44,693). For Quantization 3 the results are more consistence: for instance, Model 3 LUTs drop from 124,312 to 47,267 and DSPs from 3,165 to 0 at {8,4}, while Model 4 LUTs fall from 28,877 to 16,986 and DSPs from 1,074 to 0 at {8,4}. Overall, Q3 achieves the most aggressive area savings while Q1 is not that consistent.

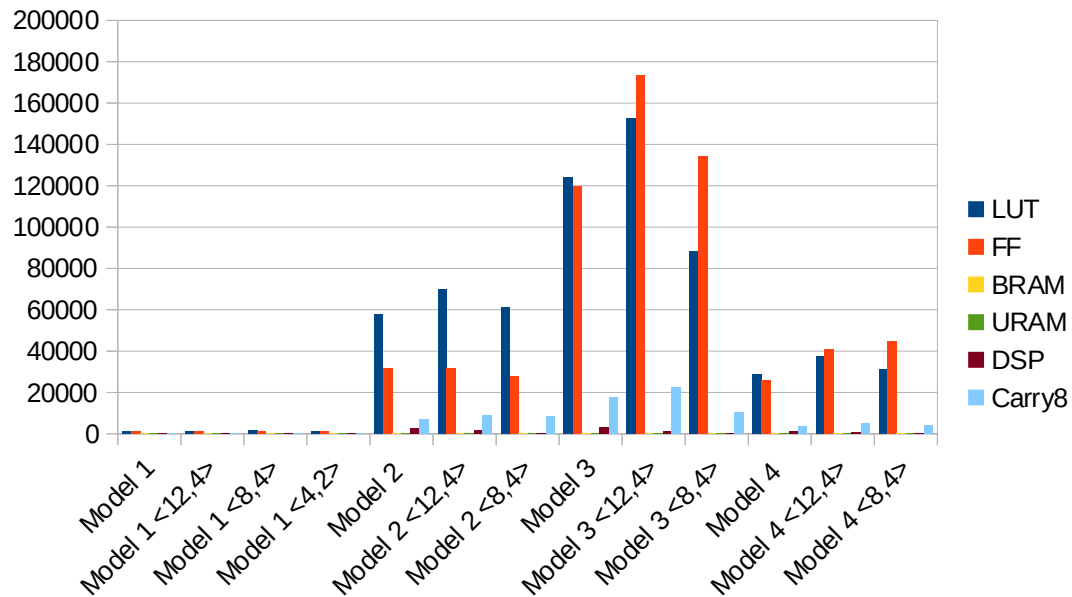


Figure 49: Utilization trade offs for Quantization 1.

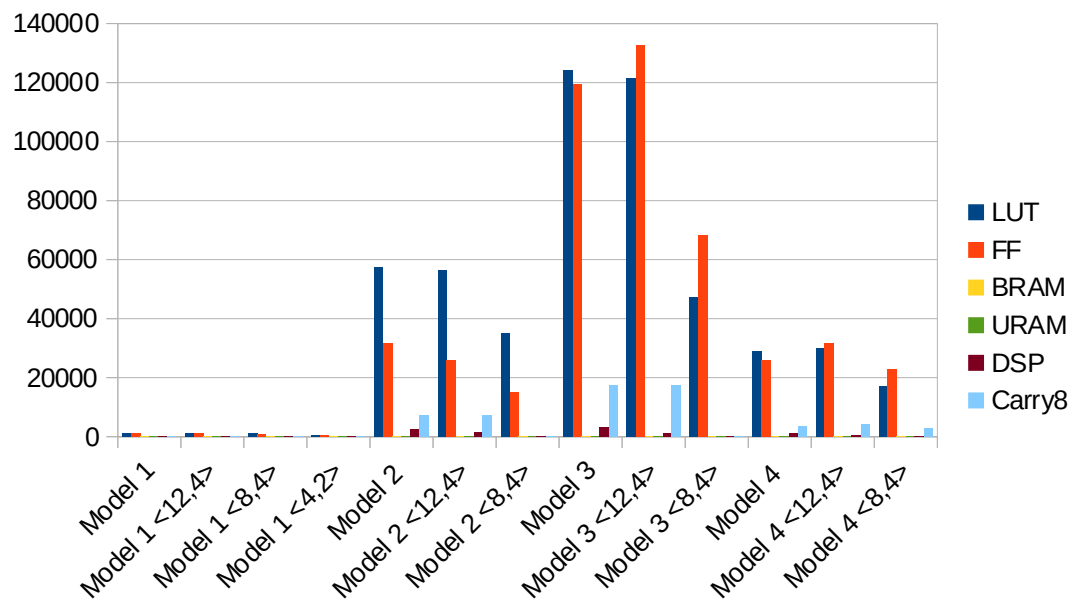


Figure 50: Utilization trade offs for Quantization 2.

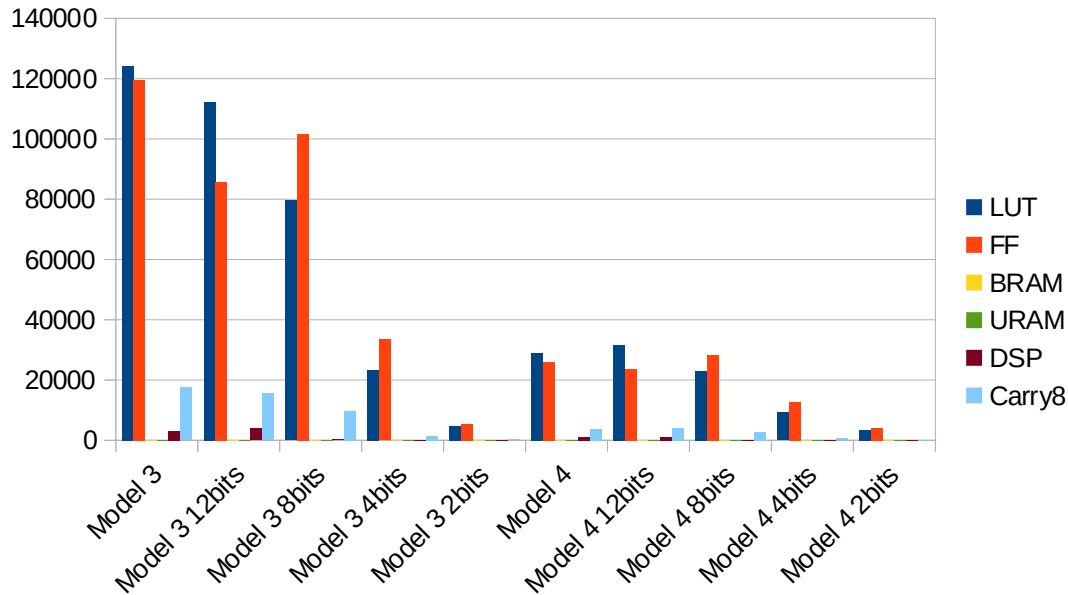


Figure 51: Utilization trade offs for Quantization 3.

These Results become more clear when we look the Figures 52 and 53. The trade-off analysis between hardware resource utilization, measured in Look-Up Tables (LUTs), and model accuracy reveals significant differences in the efficiency of the examined quantization methods. As depicted in the charts for Model 3 and Model 4, the Quantization-Aware Training (QAT) strategy, corresponding to "Path Q3" emerges as the most effective. This path forms a distinct Pareto Front, offering a gradual reduction in LUT usage with minimal impact on accuracy, especially down to the 8-bit and 4-bit levels. For instance, in Model 3, the 2-bit QAT implementation achieves an impressive 96% reduction in LUTs (from ~124k to ~4,7k), while accuracy drops by 4,6%.

Conversely, the Post-Training Quantization (PTQ) methods, corresponding to "Path Q1" and "Path Q2" prove to be significantly inferior. Their behavior is erratic and there is a sharp drop in accuracy below 70% at the 8-bit precision level. A noteworthy phenomenon is that certain PTQ implementations, such as M3_Q1_<12,4> and M4_Q1_<12,4>, lead to an increase in the required LUTs compared to the original model, rendering them counterproductive. Furthermore, the comparison between the two charts highlights the immense efficiency of pruning, as Model 4 starts from a resource baseline that is approximately 75% lower than that of Model 3.

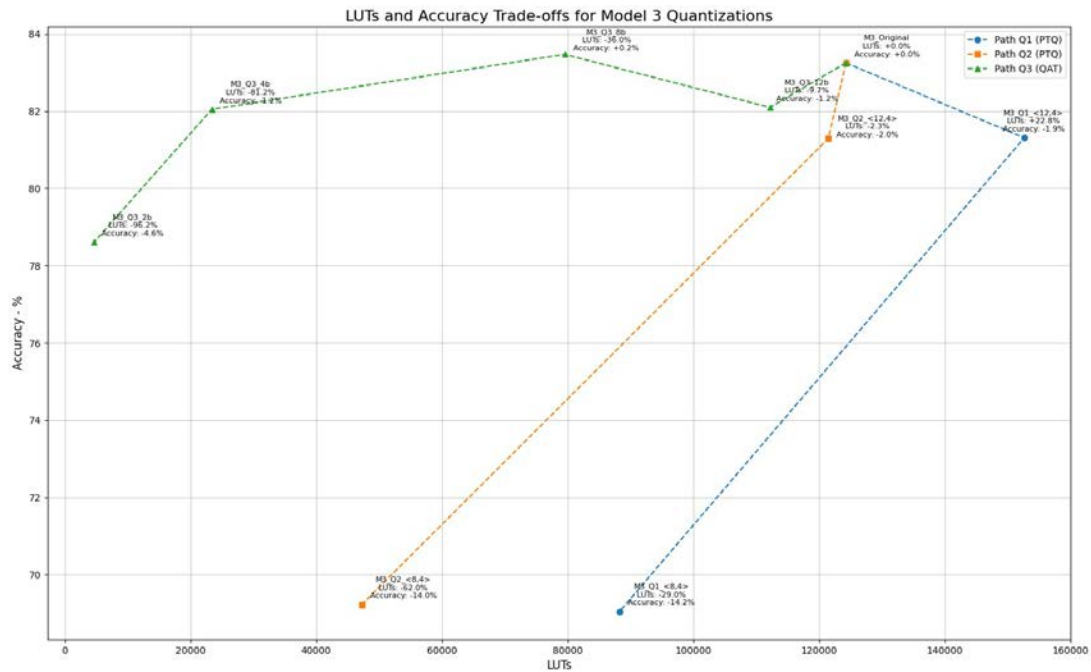


Figure 52: LUTs-Accuracy Trade-offs for Model 3.

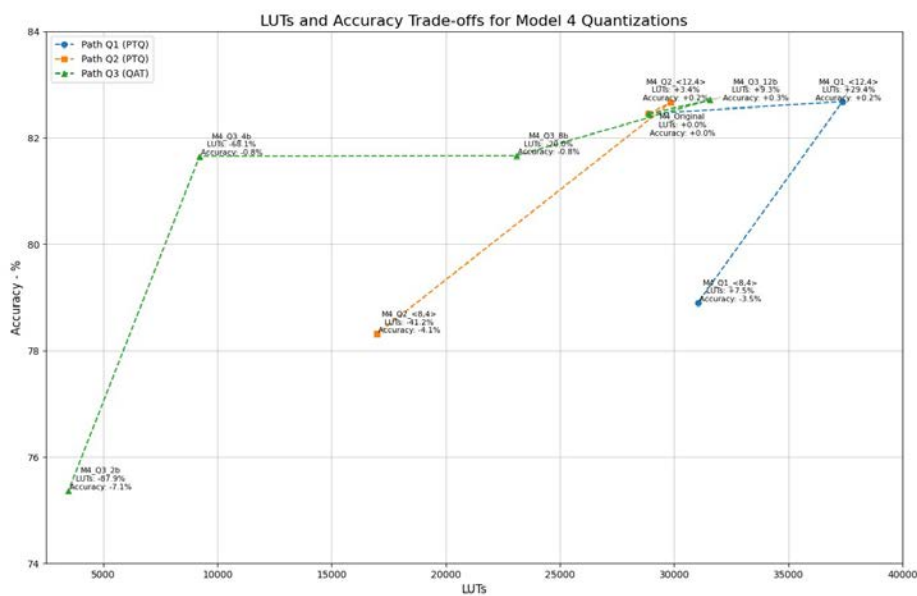


Figure 53: LUTs-Accuracy Trade-offs for Model 4.

5.2.4 Power

From the Power Report nine values were examined, Total Power, Dynamic Power, Static Power, Clock Power, Signal Power, Logic Power, BRAM Power, URAM

Power and DSP Power. For the first three values is the power from the whole design used in Vivado while for the rest is just the power used in IP Block which represent the Neural Network.

Total Power: During all quantizations as the precision was lowered so did the Total Power. On average the Q1 had 20,75% drop, Q2 had 23,45% drop and Q3 had 32,49% drop.

Dynamic Power: Dynamic Power was also lowered as smaller precision was tested during quantization. On average the Q1 had 58,48% drop, Q2 had 68,23% drop and Q3 had 89,66% drop.

Static Power: Static Power had vary small drop as smaller precision was tested during quantization. On average the Q1 had 0,28% drop, Q2 had 0,74% drop and Q3 had 1,04% drop.

Clock Power: For Clock Power there are some precision tested which results in increase Clock Power but on average the Clock Power was dropped. Specifically the Q1 had 21,13% drop, Q2 had 48,32% drop and Q3 had 90,73% drop.

Signal Power: When smaller precision was tested during the quantization the Signal Power lowered. On average the Q1 had 73,93% drop, Q2 had 88,51% drop and Q3 had 98,42% drop.

Logic Power: The Logic Power also drops as the precision drops. On average the Q1 had 76,62% drop, Q2 had 89,19% drop and Q3 had 98,24% drop.

BRAM Power: During quantization the the number of BRAMs decreased so did the BRAM and vice versa. On average the Q1 had 1,94% drop, Q2 had 2,75% drop and Q3 had 56% drop.

URAM Power: URAM Power decreased as the precision did. On average the Q1 had 18,17% drop, Q2 had 55,38% drop and Q3 had 61,22% drop. The averages were calculated using only the models which used URAM before quantization.

DSP Power: The DSP Power dropped almost to 0 for all the quantizations. Specifically on average Q1 had 99,88% drop, Q2 had 100% drop and Q3 had 100% drop. This happens because when the model is quantized to 8bit or lower the DSPs usage drops almost to 0.

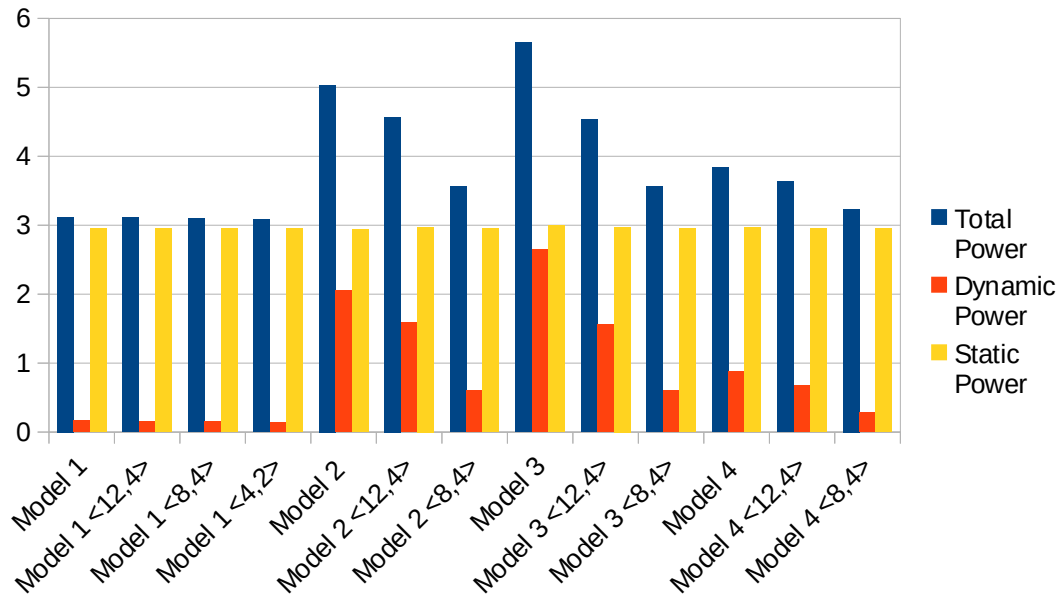


Figure 54: Total, Dynamic and Static Power trade offs for Quantization 1.

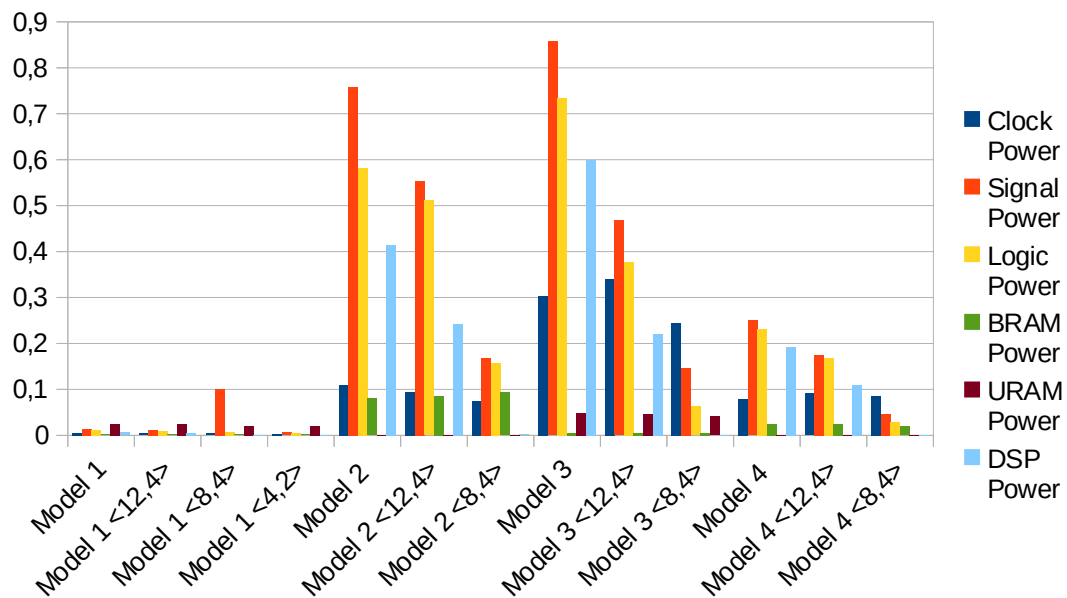


Figure 55: Clock, Signal, Logic, BRAM, URAM and DSP Power trade offs for Quantization 1.

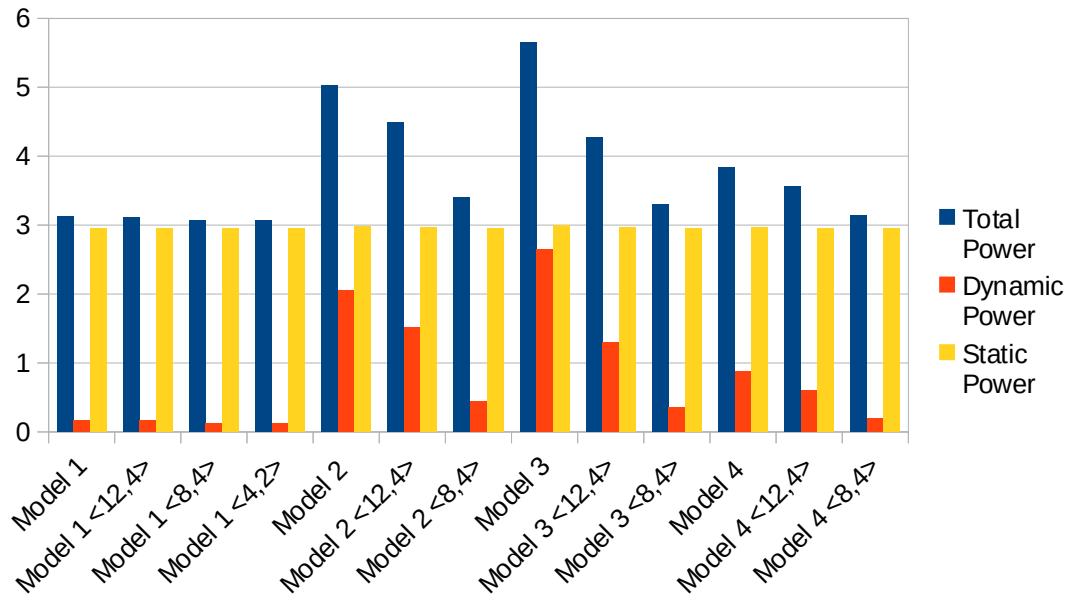


Figure 56: Total, Dynamic and Static Power trade offs for Quantization 2.

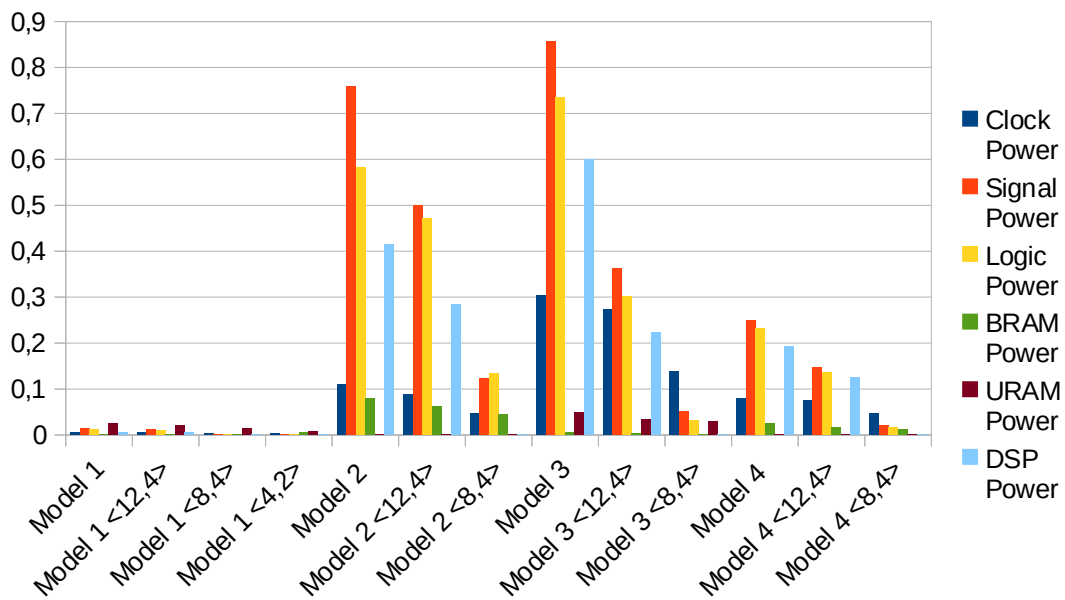


Figure 57: Clock, Signal, Logic, BRAM, URAM and DSP Power trade offs for Quantization 2.

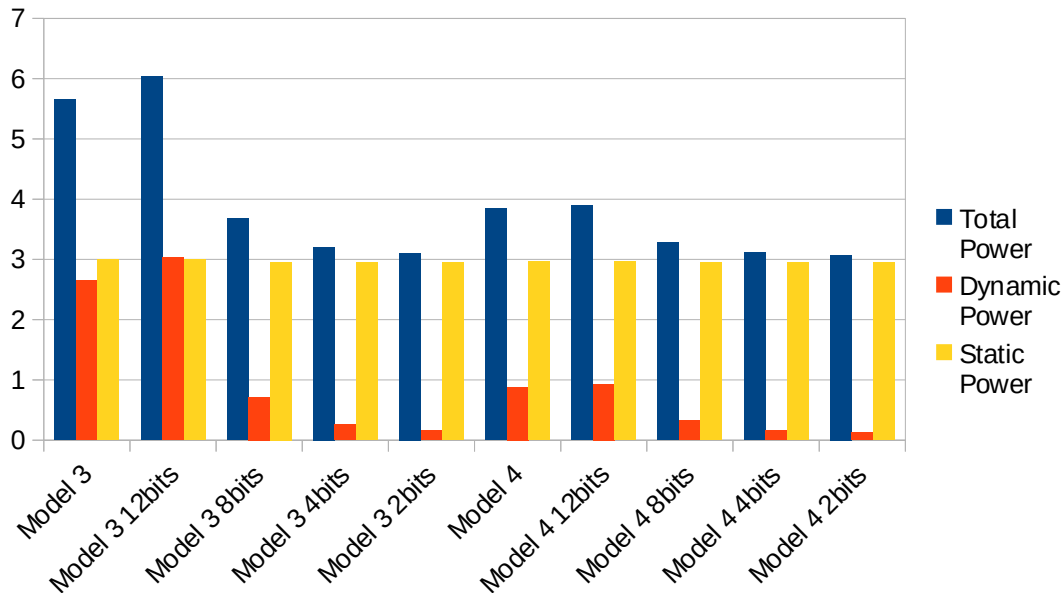


Figure 58: Total, Dynamic and Static Power trade offs for Quantization 3.

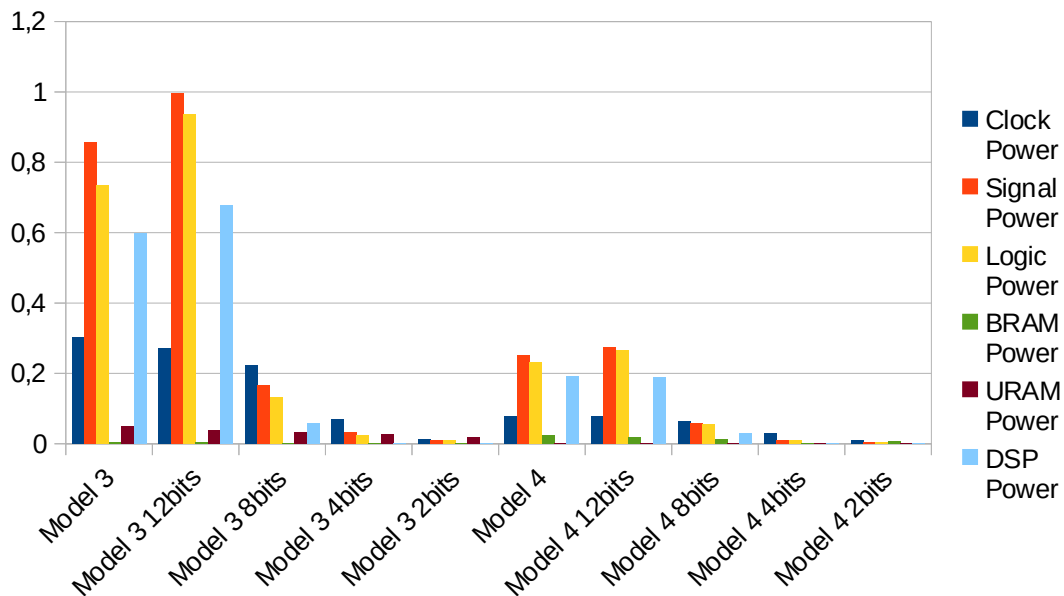


Figure 59: Clock, Signal, Logic, BRAM, URAM and DSP Power trade offs for Quantization 3.

Lower precision consistently reduces power, with the largest gains with Quantization 3. On average, Total Power -32,49%, Dynamic -89,66%, Clock -90,73%, and Signal -98,42%, Logic -98,24%, BRAM -56%, URAM -61,22% and DSP dropped to 0 -100%, while Static Power changes minimally -1,04%. For example, in Q3, Model 3's dynamic power drops from 2,654 W to 0,158 W (~94% drop) and its total power from 5,65 W to 3,106 W (~45% drop), Model 4's dynamic

power falls from 0,883 W to 0,130 W (~85% drop) and total power from 3,843 W to 3,076 W (~20% drop). Quantization 1 and Quantization 2 also show strong downward trends in Power consumption e.g. Q2 averages indicate substantial power reduction, total power -23,45% and dynamic power -68,23%.

The analysis of the trade-off between Total Power consumption and model Accuracy, shown in Figures 60 and 61, also confirm these Results. The results, visualized for both Model 3 and Model 4, show that the Quantization-Aware Training (QAT) approach, represented by "Path Q3" establishes the optimal Pareto Front. This path demonstrates a highly efficient trade-off, enabling substantial reductions in power with only a marginal loss in accuracy. For instance, the 4-bit QAT version of Model 3 achieves a remarkable power reduction of 43,2% while incurring a negligible accuracy penalty of just 1,2% compared to its baseline. This highlights the ability of QAT to produce power-efficient models without significantly compromising their predictive performance.

In contrast, the Post-Training Quantization (PTQ) strategies ("Path Q1" and "Path Q2") are shown to be significantly suboptimal. These paths are characterized by a steep decline in accuracy for a given power saving, particularly evident in the <8,4> versions which suffer a severe performance drop of over 14% and 4% for Model 3 and Model 4 respectively. Furthermore, a comparison between the two charts underscores the profound impact of pruning. The entire operational power envelope of Model 4 is shifted to much lower range than that of Model 3. For example, the M4_Original works with ~30% less Total Power consumption than the M3_Original model.

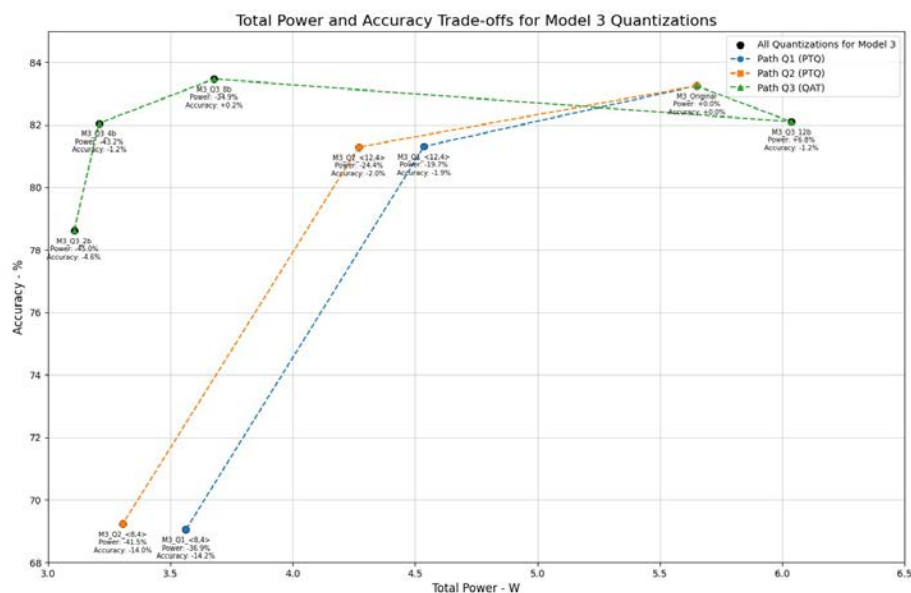


Figure 60: Total Power-Accuracy Trade-offs for Model 3.

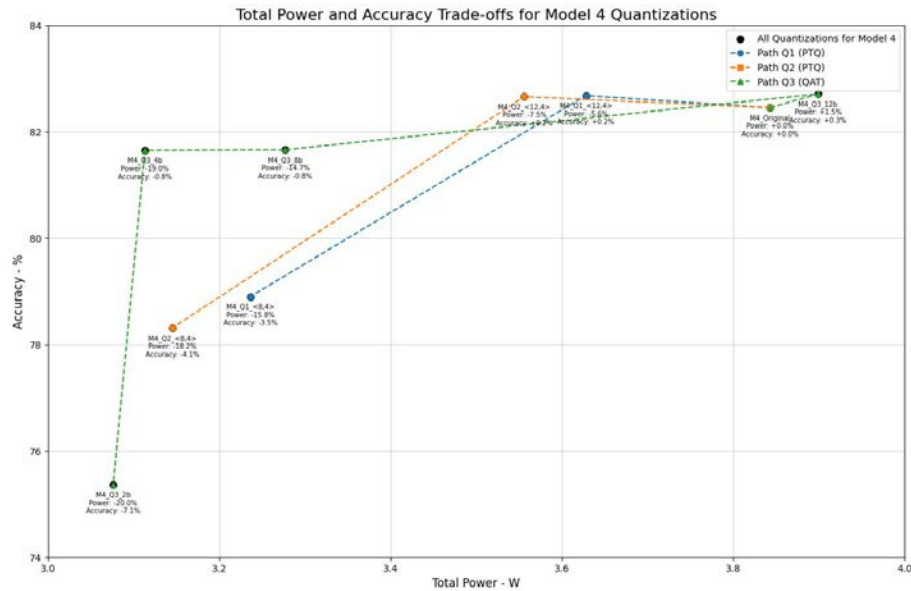


Figure 61: Total Power-Accuracy Trade-offs for Model 4.

5.2.5 Timing Improvement

As shown from the previous results the Quantization 3 manages to have the best Results for the Power and the Utilization. It is also important to see what will happen to Power and Utilization by minimizing the Clock Period since Latency is also important when the target is to make possible the implementation on NN for arrhythmia detection in devices with limited resources.

All the results shown from Chapter 5.2.2 to Chapter 5.2.4 are with Clock Period equal to 10 ns (iso-delay). Since Quantization 3 had the best results, clock improvements were done for Model 3, Model 4 and their quantized versions with Quantization 3 technique. These are the new optimized-delays for every model after testing without having any Timing Violation:

- **Model 3: 5ns**
- **Model 3 12bits: 6ns**
- **Model 3 8bits: 5ns**
- **Model 3 4bits: 3ns**
- **Model 3 2bits: 3ns**
- **Model 4: 5ns**
- **Model 4 12bits: 5ns**
- **Model 4 8bits: 4ns**
- **Model 4 4bits: 3ns**
- **Model 4 2bits: 3ns**

The results of the Timing, Utilization and Power Reports from the new Clock Periods can be seen in the tables 10-12.

Models	WNS	WHS	WPWS	Clock Period
M3 Opt	0,002	0,01	1,8	5
M3_12b Opt	0,078	0,01	2,05	6
M3_8b Opt	0,083	0,01	1,8	5
M3_4b Opt	0,003	0,01	0,8	3
M3_2b Opt	0,084	0,007	0,8	3
M4 Opt	0,023	0,006	1,96	5
M4_12b Opt	0,109	0,011	1,96	5
M4_8b Opt	0,014	0,01	1,46	4
M4_4b Opt	0,064	0,01	0,96	3
M4_2b Opt	0,235	0,002	0,96	3

Table 10: Timing Results after Clocking Optimization.

Models	LUT	FF	BRAM	URAM	DSP	Carry8
M3 Opt	124318	121037	5,5	12	3165	17538
M3_12b Opt	112215	85718	4	10	3875	15526
M3_8b Opt	79551	101637	3	8	311	9703
M3_4b	23449	33507	2	7	0	1410
M3_2b	4668	5517	3,5	5	0	241
M4	28876	26008	28,5	0	1074	3734
M4_12b	31568	23580	23	0	1066	4058
M4_8b	23105	28332	18,5	0	178	2773
M4_4b	9222	12692	14	0	0	626
M4_2b	3481	4071	10,5	0	0	171

Table 11: Utilization Results after Clocking Optimization.

Models	Total Power	Dynamic Power	Static Power	Clock Power	Signal Power	Logic Power	BRAM Power	URAM Power	DSP Power
M3 Opt	8,488	5,435	3,052	0,57	1,998	1,466	0,009	0,099	1,184
M3_12b Opt	8,37	5,32	3,05	0,448	1,989	1,576	0,006	0,064	1,131
M3_8b Opt	4,297	1,327	2,97	0,43	0,341	0,262	0,006	0,065	0,117
M3_4b Opt	3,578	0,621	2,956	0,235	0,106	0,074	0,006	0,092	0
M3_2b Opt	3,249	0,299	2,95	0,052	0,034	0,034	0,011	0,063	0
M4 Opt	4,655	1,679	2,976	0,156	0,517	0,464	0,05	0	0,387
M4_12b Opt	4,763	1,785	2,978	0,157	0,562	0,531	0,034	0	0,377

M4_8b Opt	3,617	0,66	2,957	0,161	0,144	0,135	0,032	0	0,077
M4_4b Opt	3,268	0,318	2,95	0,103	0,039	0,034	0,032	0	0
M4_2b Opt	3,147	0,199	2,948	0,04	0,018	0,015	0,022	0	0

Table 12: Power Results after Clocking Optimization.

5.2.5.1 Timing Results from Clock Optimization

By looking the results from the Timing Report it is clear that when the models are quantized to smaller number of bits, smaller clock period can be achieved. The only exception is the Model 3 at 12 bit which got 6ns Clock Period while Model 3 at 4 and 2 bit got the smallest Clock Period at 3ns. Also Model 4 got the smallest Clock Period at 4 and 2 bit. WNS, WHS and WPWS got lower for all the models. The trade offs can be seen in Figures 54 and 55. The Opt referred to the Clock Period Optimized models.

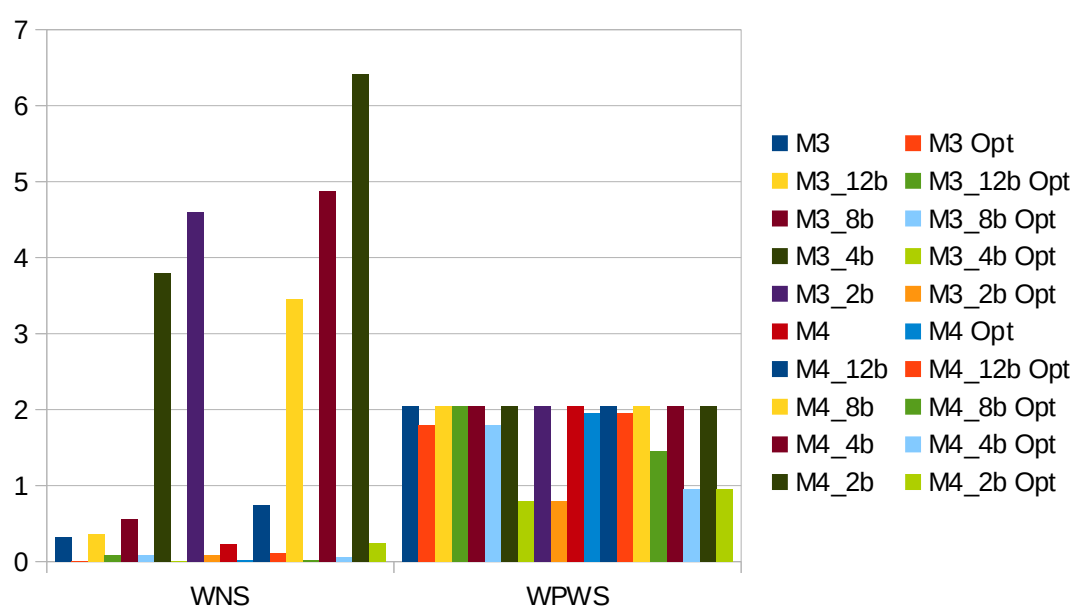


Figure 62: WNS and WPWS trade offs from Clock Optimization.

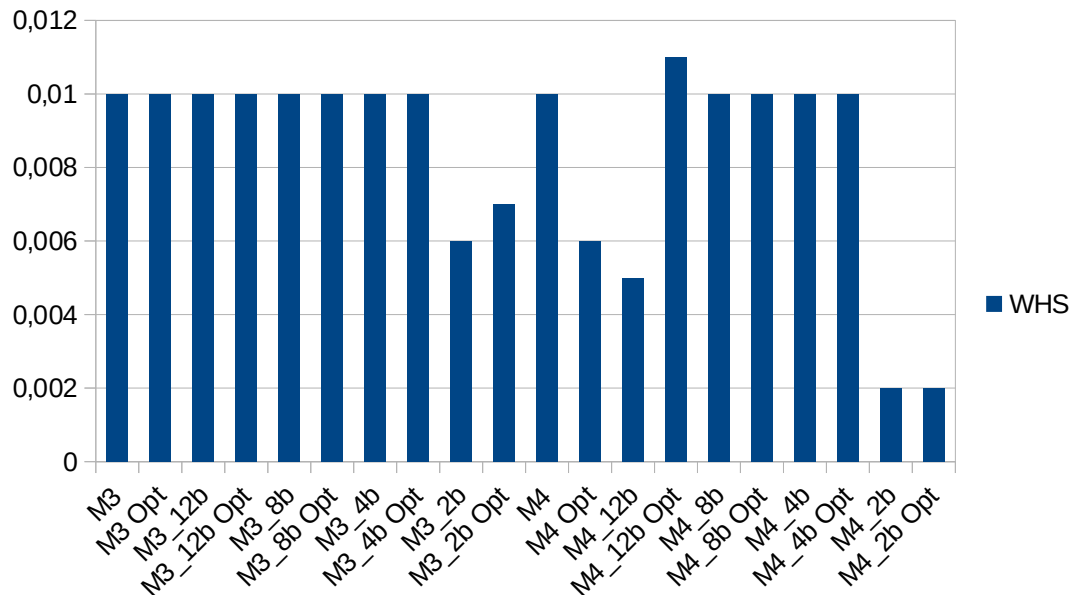


Figure 63: WHS trade offs for Clock Optimization.

We observe that pushing below the 10 ns iso-delay delivers 40–70% shorter clock periods depending on precision: for Model 3, 6 ns at 12-bit (–40%), 5 ns at 8-bit (–50%), and 3 ns at 4 and 2 bit (–70%), for Model 4, 5 ns at 12-bit (–50%), 4 ns at 8-bit (–60%), and 3 ns at 4 and 2 bit (–70%). However, M3 at 12-bit had the smallest improvement (6 ns), while lower precisions consistently enable the most aggressive clocks. All optimized builds keep lower their slacks but they kept the positive (e.g., M3_4b opt: WNS 0.003 ns / WHS 0.010 ns / WPWS 0.8 ns; M4_4b opt: 0.064 ns / 0.010 ns / 0.958 ns), confirming no timing violations as frequency increases.

5.2.5.2 Utilization Results from Clock Optimization

The optimization of the Clock Period had very small effect to the Utilization Reports of the models. Specifically BRAM, URAM, DSPs and CARRY8s weren't affected at all. Also FFs weren't affected by this change expect of the Model 3 before the quantization, where the FF's number went slightly up from 119.517 to 121.037. Only the number of LUTs was changed every time and had a small increase except of the Model 4 before quantization, Model 4 at 12 bit and Model 4 at 2 bit. The trade offs for the LUTs can be seen on the Table 13, where the Opt referred to the Clock Period Optimized models..

Models	LUT
M3	124312
M3 Opt	124318
M3_12b	112214
M3_12b Opt	112215

M3_8b	79545
M3_8b Opt	79551
M3_4b	23414
M3_4b Opt	23449
M3_2b	4662
M3_2b Opt	4668
M4	28877
M4 Opt	28876
M4_12b	31570
M4_12b Opt	31568
M4_8b	23097
M4_8b Opt	23105
M4_4b	9219
M4_4b Opt	9222
M4_2b	3482
M4_2b Opt	3481

Table 13: LUTs trade offs from Clock Optimization.

5.2.5.3 Power Results from Clock Optimization

The biggest change from Clock Optimization occurred to the Power consumption where all the values raised. On the Table 14 there is the percent raised of the Total Power for all the models.

Models	Total Power	Raise
M3 Opt	8,488	50,23%
M3_12b Opt	8,37	38,64%
M3_8b Opt	4,297	16,82%
M3_4b Opt	3,578	11,53%
M3_2b Opt	3,249	4,60%
M4 Opt	4,655	21,12%
M4_12b Opt	4,763	22,15%
M4_8b Opt	3,617	10,37%
M4_4b Opt	3,268	4,97%
M4_2b Opt	3,147	2,30%

Table 14: Total Power and the Percent Raise of Clock Optimized Models.

On the Table 15 there are the Results for all the Values of the Power Report and on the Figures 56 and 56 there are the trade offs that the Clock Optimization had to the power consumption of the models.

Models	Total Power	Dynamic Power	Static Power	Clock Power	Signal Power	Logic Power	BRAM Power	URAM Power	DSP Power
M3 Opt	8,488	5,435	3,052	0,57	1,998	1,466	0,009	0,099	1,184

M3_12b										
Opt	8,37	5,32	3,05	0,448	1,989	1,576	0,006	0,064	1,131	
M3_8b										
Opt	4,297	1,327	2,97	0,43	0,341	0,262	0,006	0,065	0,117	
M3_4b										
Opt	3,578	0,621	2,956	0,235	0,106	0,074	0,006	0,092	0	
M3_2b										
Opt	3,249	0,299	2,95	0,052	0,034	0,034	0,011	0,063	0	
M4 Opt	4,655	1,679	2,976	0,156	0,517	0,464	0,05	0	0,387	
M4_12b										
Opt	4,763	1,785	2,978	0,157	0,562	0,531	0,034	0	0,377	
M4_8b										
Opt	3,617	0,66	2,957	0,161	0,144	0,135	0,032	0	0,077	
M4_4b										
Opt	3,268	0,318	2,95	0,103	0,039	0,034	0,032	0	0	
M4_2b										
Opt	3,147	0,199	2,948	0,04	0,018	0,015	0,022	0	0	

Table 15: Results of the Power Report after the Clock Optimization.

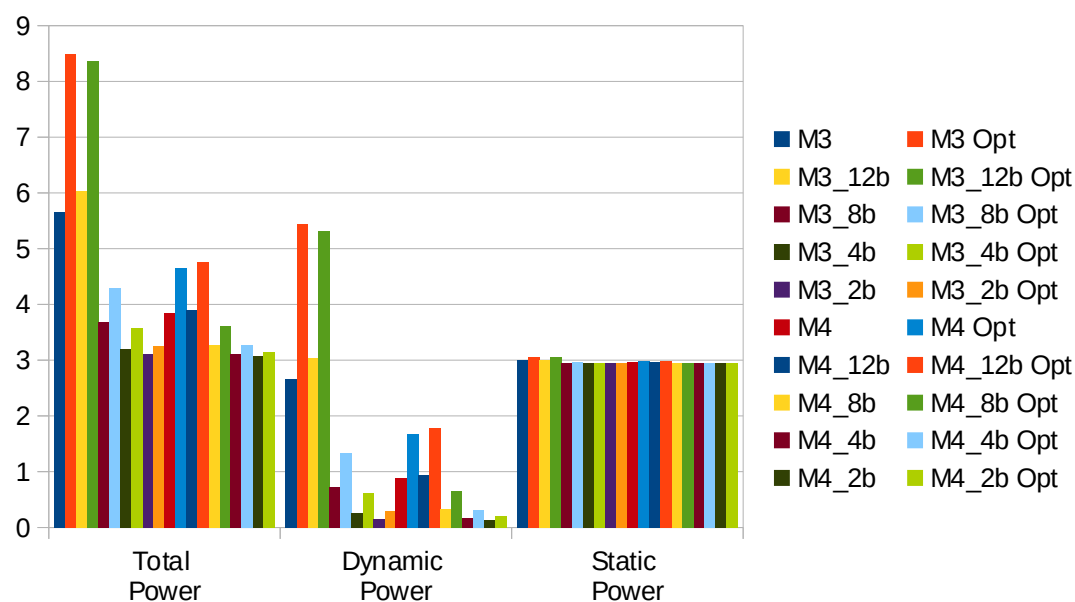


Figure 64: Total, Dynamic and Static Power trade offs from the Clock Optimization.

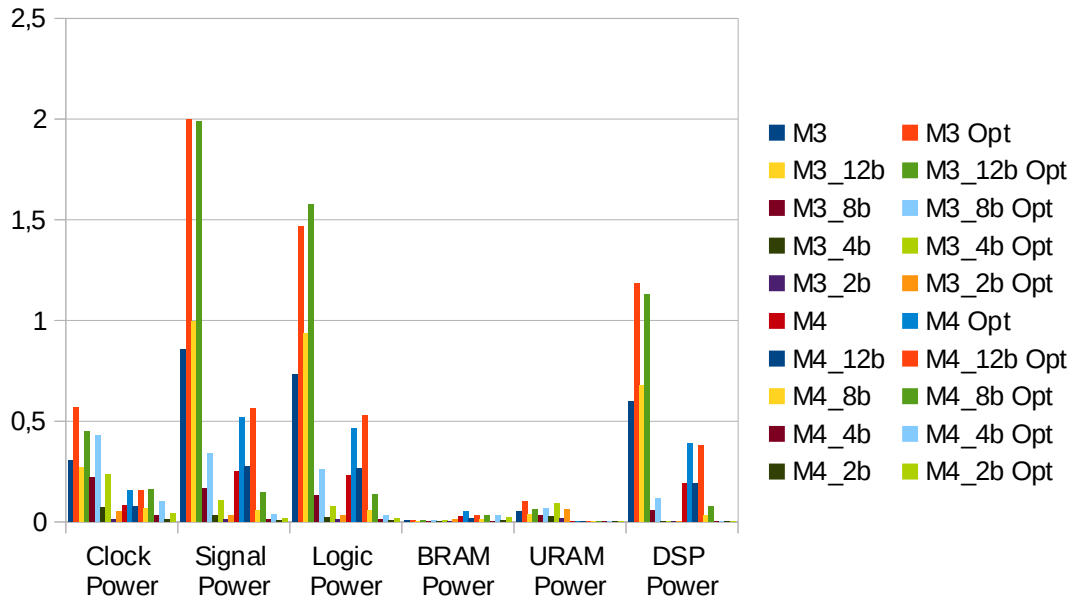


Figure 65: Clock, Signal, Logic, BRAM, URAM and DSP Power trade off of the Clock Optimization.

The results in Table 14 demonstrate that a reduction in the clock period leads to an increase in power consumption. This effect is smaller when the model is quantized to smaller precisions. For instance Model 3 at 12-bits clock period was lowered 40% and its total power increased by 38,64%, while on 2-bits clock period was lowered 70% and total power increased just 4,6%. The same effect is observed for Model 4. In 12-bits the clock period was lowered 50% and total power increased 22,15%, while in 2-bits clock period was lowered 70% and total power increased 2,3%.

When we look trade-offs between Total Power consumption and Optimized Delay (Figure 66), compared with the Original Optimized models we can see positive results. The analysis of the two optimized model architectures, as illustrated in the diagram, clearly demonstrates the benefits of pruning and quantization. The operational path of Model 4 is situated in a significantly more efficient region of the design space compared to Model 3, confirming the immense impact of pruning on power reduction. For the same initial 5 ns latency, the baseline M4_Opt model consumes approximately 45% less power than the M3_Opt model. Furthermore, for both architectures, the application of quantization (moving from top-right to bottom-left along each path) successfully reduces both power and delay simultaneously. The most optimal solution is achieved by the M4_2b_Opt configuration, which leverages both pruning and aggressive 2-bit quantization to deliver the lowest latency (3 ns) at the lowest power consumption (~3,1 W), establishing it as the most efficient design point among all tested variants.

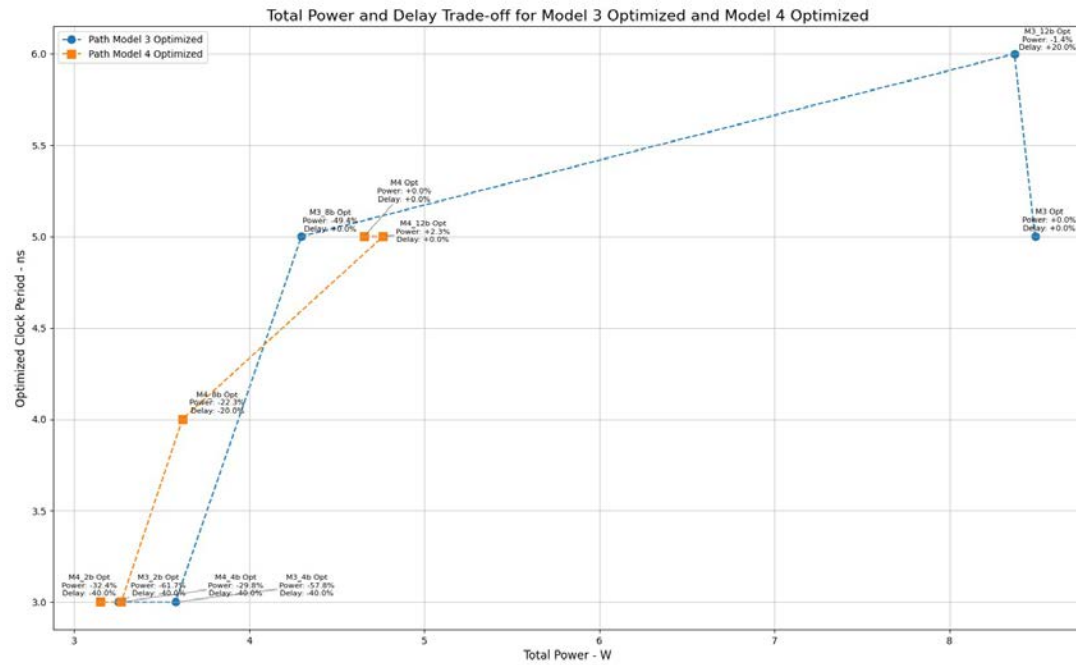


Figure 66: Total Power and Delay Trade-offs for Model 3 Optimized and Model 4 Optimized.

Chapter 6: Conclusions

The implementation of Neural Networks for cardiac arrhythmia detection or for other cardiovascular diseases, is very important since they are the leading cause of mortality according to World Health Organization^[40]. This implementation allows the detection of CVDs early, allowing the patient to take immediately the necessary measurements.

In order to make the implementation of such Networks possible in portable devices, PTQ and QAT are tested. Also the combination of pruning with these technique is used. Specifically, just by applying pruning with 50% sparsity on the Model 3, the biggest model tested, there was 31,98% drop in the Total Power. At the same time, there was also a huge drop in the components used, LUTs from 124.312 dropped to 28.877 and FFs from 119.517 dropped to 26.008.

From the quantizations used, QAT technique perform much better from the other two PTQ techniques tested. Specifically, the average Total Power drop for Q1 and Q2 was 20,75% and 23,45% while for Q3 it was 32,49%. Also resources utilization drop was higher when the Q3 was used, sometimes more than doubled.

Applying pruning with 50% sparsity and QAT combined, can achieve even better results. Specifically when pruning with 50% sparsity and QAT to 2bit was applied to the Model 3, the Total Power had dropped by 45,56%. The components used also dropped significantly. LUTs dropped 97,2%, FFs dropped 96,59%, Carry8 dropped 99,02% and URAM and DPSs dropped 100%. Only BRAM increased by 90,91% but this doesn't mean increased demand in memory because URAM much larger memory block. After combining these two technique there was an accuracy drop of 7,88%. Nevertheless if QAT is performed with 4 bit, accuracy drops only 1,6% and the results are also great.

When it comes to lowering the latency of the Neural Networks there is a trade off with the Power Consumption that need to be considered. Quantizing the model minimize this trade off but it doesn't eliminate it. So it important to target the latency that cover the need but has the least increase on the Power Consumption. Also Quantization allow to achieve lower latency without any Timing Violation.

Despite these promising outcomes, there are also limitations. In order to be realistic the integration of Neural Networks for CVDs detection in resource constrained devices, models with more than 99% accuracy must be used. These models in order to achieved this accuracy are usually very large with more than 1.000.000 parameters. The method used by HLS4ML for turning such models to HDL is very memory demanding making it impossible to be tested with the normal hardware used in this thesis. However, it is anticipated that applying this optimization methodology to larger, implementable models with higher accuracy would yield similarly positive results in resource reduction.

In conclusion, the techniques used in this thesis showed promising results in simplification of Neural Networks for arrhythmia detections without the need of huge accuracy loss. But it is important to develop less memory demanding algorithms for converting Neural Networks in HDL, in order to be easier to test compression techniques in larger models with better accuracy. In this way the implementation of reliable Neural Networks for CVDs detection in resource constrained devices will be realistic.

References

- [1] Cahyo L., & Astuti S. D. (2023). Early Detection of Health Problems through Artificial Intelligence (Ai) Technology in Hospital Information Management: A Literature Review Study. *Journal of Medical and Health Studies*, 4(3), 37–42.
- [2] Cheng Y., Wang D., Zhou P., & Zhang T. (2020). A survey of model compression and acceleration for deep neural networks. *arXiv*.
- [3] Sze V., Chen Y.-H., Yang T.-J., & Emer J. (2017). Efficient processing of deep neural networks: A tutorial and survey. *arXiv*.
- [4] Tasci M., Istanbulu A., Tumen V., & Kosunalp S. (2025). FPGA-QNN: Quantized neural network hardware acceleration on FPGAs. *Applied Sciences*, 15(2), 688.
- [5] Goyal R., Vanschoren J., van Acht V., & Nijssen S. (2021). Fixed-point quantization of convolutional neural networks for quantized inference on embedded platforms. *arXiv*.
- [6] Delgado De Molina Rius, A. (2023). Chapter 1: Foundations of artificial intelligence and machine learning. *Artificial intelligence in finance: Challenges, opportunities and regulatory developments* (pp. 2–18).
- [7] Samuel, A. L. (1959). Some studies in machine learning using the game of checkers. *IBM Journal of Research and Development*, 3(3), 210-229.
- [8] Mitchell, T. M. (1997). *Machine Learning*. McGraw-Hill.
- [9] Sindhu V, Nivedha S, Prakash M (February 2020). "An Empirical Science Research on Bioinformatics in Machine Learning". *Journal of Mechanics of Continua and Mathematical Science*.
- [10] Jordan M. I., & Mitchell T. M. (2015). Machine learning: Trends, perspectives, and prospects. *Science*, 349(6245), 255–260.
- [11] Haykin, S. (2009). *Neural Networks and Learning Machines* (3rd ed.). Pearson.
- [12] Kotecha, K., & Dhemale, S. L. (2017). Neural Networks – State of Art, Brief History, Basic Models and Architecture. In K. Kotecha, R. K. Ovaeswaran, & S. N. Sivanandam (Eds.), *Computational Intelligence and Soft Computing* (pp. 3–14).
- [13] Ansari Y., Mourad O., Qaraq Kh., & Serpedin E. (2023). Deep learning for ECG Arrhythmia detection and classification: an overview of progress for period 2017–2023. *Frontiers in Physiology*, 14, 1246746.

- [14] Martínez-Sellés, M., & Marina-Breyse, M. (2023). Current and Future Use of Artificial Intelligence in Electrocardiography. *Journal of Cardiovascular Development and Disease*, 10(4), 175.
- [15] Bayani A., & Kargar M. (2024). LDCNN: A new arrhythmia detection technique with ECG signals using a linear deep convolutional neural network. *Physiological Reports*, 12(17).
- [16] Khan M. A., & Kim Y. (2021). Cardiac arrhythmia disease classification using LSTM deep learning approach. *Computers, Materials & Continua*, 67(1), 427–443.
- [17] Sun, J. (2023). Automatic cardiac arrhythmias classification using CNN and attention-based RNN network. *Healthcare Technology Letters*, 10(3), 53–61
- [18] Saranya K., Karthikeyan U., Kumar A. S., Salau A. O., & Tin T. T. (2025). DenseNet-ABiLSTM: Revolutionizing multiclass arrhythmia detection and classification using hybrid deep learning approach leveraging PPG signals. *International Journal of Computational Intelligence Systems*, 18, article 33.
- [19] Kim D., Lee K. R., Lim D. S., Lee K. H., Lee J. S., Kim D. Y., & Sohn C. B. (2025). A novel hybrid CNN-transformer model for arrhythmia detection without R-peak identification using Stockwell transform. *Scientific Reports*, 15, Article 7817.
- [20] Elhanashi A., Dini P., Saponara S., & Zheng Q. (2024). Advancements in TinyML: Applications, limitations, and impact on IoT devices. *Electronics*, 13(17), 3562.
- [21] Sazonov, E., & Daoud, W. A. (2021). Grand challenges in wearable electronics. *Frontiers in Electronics*, 2, Article 668619.
- [22] Liu D., Zhu Y., Liu Z., Liu Y., Han C., Tian J., Li R., & Yi W. (2025). A survey of model compression techniques: past, present, and future. *Frontiers in Robotics and AI*, 12.
- [23] Han, S., Pool, J., Tran, J., and Dally, W. J. (2015). Learning both weights and connections for efficient neural networks. *Proc. 28th Int. Conf. Neural Inf. Process. Syst.* -1, 1135–1143.
- [24] Mocanu, D. C., Mocanu, E., Stone, P., Nguyen, P. H., Gibescu, M., and Liotta, A. (2018). Scalable training of artificial neural networks with adaptive sparse connectivity inspired by network science. *Nat. Commun.* 9, 2383.
- [25] Alqahtani, A., Xie, X., Essa, E., and Jones, M. W. (2021). Neuron-based network pruning based on majority voting. *2020 25th Int. Conf. Pattern Recognit. (ICPR)*, 3090–3097.
- [26] Srinivas, S., and Babu, R. V. (2015). “Data-free parameter pruning for deep neural networks,” in *British Machine Vision Conference*, 31.1–31.12.

- [27] Mariet, Z., and Sra, S. (2016). “Diversity networks: neural network compression using determinantal point processes,” in International Conference on Learning Representations.
- [28] Li, H., Kadav, A., Durdanovic, I., Samet, H., and Graf, H. P. (2017). “Pruning filters for efficient convnets,” in International Conference on Learning Representations.
- [29] Hu, H., Peng, R., Tai, Y.-W., and Tang, C.-K. (2016). Network trimming: a data-driven neuron pruning approach towards efficient deep architectures. ArXiv.
- [30] hls4ml Documentation (2025)[Software]. Available at: <https://fastmachinelearning.org/hls4ml/>
- [31] Cristiano Aguiar (2020). arrhythmia-detection [Software]. GitHub <https://github.com/csaguiar/arrhythmia-detection/tree/master?tab=readme-ov-file>
- [32] Goldberger, A., Amaral, L., Glass, L., Hausdorff, J., Ivanov, P. C., Mark, R., ... & Stanley, H. E. (2000). PhysioBank, PhysioToolkit, and PhysioNet: Components of a new research resource for complex physiologic signals. Circulation [Online]. 101 (23)
- [33] Moody GB, Mark RG. The impact of the MIT-BIH Arrhythmia Database. IEEE Eng in Med and Biol 20(3):45-50 (May-June 2001).
- [34] TensorFlow Developers. (2023). TensorFlow: An end-to-end open source machine learning platform (Version [2.15.1]) [Software]. Available at <https://www.tensorflow.org/>
- [35] Keras Developers. (2023). Keras: A superpower for ML developers (Version [2.15.0]) [Software]. Available at <https://keras.io/>
- [36] TensorFlow Developers. (2023). TensorFlow Model Optimization Toolkit (Version [0.7.5]) [Software]. Available at https://www.tensorflow.org/model_optimization
- [37] Xilinx Inc. (2023). Vitis High-Level Synthesis (Version [2023.2]) [Software]. Available at: <https://www.xilinx.com/products/design-tools/vitis/vitis-hls.html>
- [38] QKeras Developers. (2021). QKeras (Version [0.9.0]) [Software]. Available at: <https://github.com/google/qkeras>
- [39] Xilinx Inc. (2023). Vivado Design Suite (Version [2023.2]) [Software]. Available at: <https://www.xilinx.com/products/design-tools/vivado.html>
- [40] World Health Organization. (2025). Cardiovascular diseases (CVDs). Available at: [https://www.who.int/news-room/fact-sheets/detail/cardiovascular-diseases-\(cvds\)](https://www.who.int/news-room/fact-sheets/detail/cardiovascular-diseases-(cvds))

- [41] Kotsiantis, S. B. (2007). Supervised machine learning: A review of classification techniques. *Informatica*, 31(3), 249–268.
- [42] Goodfellow, I., Bengio, Y., & Courville, A. (2016). *Deep Learning*. MIT Press.
- [43] Xu, R., & Wunsch, D. (2005). Survey of clustering algorithms. *IEEE Transactions on Neural Networks*, 16(3), 645–678.
- [44] Aggarwal, C. C., Reddy, C. K. (2013). *Data clustering: Algorithms and applications*. CRC Press.
- [45] Sutton, R. S., & Barto, A. G. (2018). *Reinforcement learning: An introduction* MIT Press.
- [46] LeCun, Y., Bengio, Y., & Hinton, G. (2015). Deep learning. *Nature*, 521(7553), 436–444.
- [47] Venieris, S. I., & Bouganis, C. S. (2018). Toolflows for mapping convolutional neural networks on FPGAs: A survey. *ACM Computing Surveys*, 51(3), 1–39.
- [48] Nurvitadhi, E., Venkatesh, G., Sim, J., Marr, D., Huang, R., Hock, J. O. G., Liew, Y. T., Srivatsan, K., Moss, D., & Subhaschandra, S. (2017). Can FPGAs beat GPUs in accelerating next-generation deep neural networks? In *FPGA '17: Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays* (pp. 5–14).
- [49] Chowdhury, A., Chowdhury, M. H., Das, D., Ghosh, S., & Cheung, R. C. C. (2024). FPGA Implementation of PPG-Based Cardiovascular Diseases and Diabetes Classification Algorithm. *Arabian Journal for Science and Engineering*, 49, 16697–16709.
- [50] Liu, Z., Ling, X., Zhu, Y., & Wang, N. (2025). FPGA-based 1D-CNN accelerator for real-time arrhythmia classification. *Journal of Real-Time Image Processing*, 22, 66.