Embedded Machine Learning

Emphasis on Hardware Accelerators and Approximate Computing for Tactile Data Processing

Hamoud Y.Younes

Supervisor: Prof. Maurizio Valle

Co-supervisors: Dr. Ali Ibrahim and Dr. Mostafa Rizk

Department of Electrical, Electronics and Telecommunication Engineering
and Naval Architecture (DITEN)
University of Genoa

This dissertation is submitted for the degree of
Doctor of Philosophy

COSMIC LAB November 2021

I would like to dedicate this dissertation to everyone that crossed my path in this life...family, friends, and strangers . . .

Acknowledgements

I would like to express my deep appreciation to Prof. Maurizio Valle, Dr. Ali Ibrahim, and Dr. Mostafa Rizk for their supervision, guidance, and support throughout the course of Ph.D. During the pandemic, everyone kept offering their help and technical knowledge to complete the novel research presented in this dissertation.

A special thanks to the university of Genoa that gave me the chance to pursue my doctoral studies and participate in the top European conferences and summer schools.

Abstract

Machine Learning (ML) a subset of Artificial Intelligence (AI) is driving the industrial and technological revolution of the present and future. We envision a world with smart devices that are able to mimic human behaviour (sense, process, and act) and perform tasks that at one time we thought could only be carried out by humans. The vision is to achieve such level of intelligence with affordable, power-efficient, and fast hardware platforms. However, embedding machine learning algorithms in many application domains such as the internet of things (IoT), prostheses, robotics, and wearable devices is an ongoing challenge. A challenge that is controlled by the computational complexity of ML algorithms, the performance/availability of hardware platforms, and the application's budget (power constraint, real-time operation, etc.). In this dissertation, we focus on the design and implementation of efficient ML algorithms to handle the aforementioned challenges. First, we apply Approximate Computing Techniques (ACTs) to reduce the computational complexity of ML algorithms. Then, we design custom Hardware Accelerators to improve the performance of the implementation within a specified budget. Finally, a tactile data processing application is adopted for the validation of the proposed exact and approximate embedded machine learning accelerators.

The dissertation starts with the introduction of the various ML algorithms used for tactile data processing. These algorithms are assessed in terms of their computational complexity and the available hardware platforms which could be used for implementation. Afterward, a survey on the existing approximate computing techniques and hardware accelerators design methodologies is presented. Based on the findings of the survey, an approach for applying algorithmic-level ACTs on machine learning algorithms is provided. Then three novel hardware accelerators are proposed: (1) k-Nearest Neighbor (kNN) based on a selection-based sorter, (2) Tensorial Support Vector Machine (TSVM) based on Shallow Neural Networks, and (3) Hybrid Precision Binary Convolution Neural Network (BCNN). The three accelerators offer a real-time classification with monumental reductions in the hardware resources and power consumption compared to existing implementations targeting the same tactile data processing application on FPGA. Moreover, the approximate accelerators maintain a high classification accuracy with a loss of at most 5%.

Table of contents

List of figures

List of tables

Chapter 1

Introduction

The skin is the largest organ in the human body which houses receptors that sense touch. Humans sense of touch allows them to receive information about their environment, making it important for sensory perception. Touch receptors in the skin inform the brain about tactile or touch sensations. The receptors transform a response (e.g. chemical, thermal or mechanical) into electrical signals. The signals travel along axons (the elongated portion of the neuron), which form a path along which messages travel to different areas of the brain that receive and interpret them. In the brain, sensations are interpreted using a complex intelligent architecture fed on previous experiences and the properties of the receptors. With the advancements of technology and engineering, humans have been trying to mimic such capabilities through an artificial model referred to as Electronic Skin (e-skin).

An electronic skin is usually composed of distributed tactile sensors integrated with an embedded electronic system for tactile data decoding. Meaningful information e.g. texture classification and pattern recognition can be decoded from tactile data by employing Machine Learning (ML) algorithms. Computations using embedded machine learning algorithms may enable the electronic skin system to be used in various application domains such as wearable, Internet of Things, prosthetic and robotics. However, embedding machine learning algorithms is constrained by the high computational complexity of such algorithms. Moreover, the amount of data processed by machine learning is increasing exponentially [1]. On the contrary, the processing resources are limited especially with the chip shortage in the last few years [2]. This poses challenges relevant to the requirements of real-time execution and low power/energy consumption when targeting portable wearable systems due to their limitation in terms of resources and energy budget. Nonetheless, machine learning is one of many application domains that has intrinsic tolerance to inaccuracy. These applications are mostly not about calculating a precise numerical answer; instead, "correctness" is defined as providing an outcome that is good enough, or of sufficient quality

to achieve an acceptable application performance [3]. In this perspective, this dissertation aims at providing efficient implementations of embedded machine learning algorithms for tactile data processing. The core strategy behind delivering efficient implementations is the use of *"Hardware Accelerators"* and *"Approximate Computing"* to accelerate demanding portions of ML algorithms and to adequately reduce the algorithms computational complexity respectively.

1.1 Objectives and Contributions

The main objective of this dissertation is to design efficient implementations of machine learning algorithms for tactile data processing. The implementations should offer real-time processing with a time latency less than 400 ms [4] with as much reduced hardware area and energy consumption as possible compared to existing solutions. To achieve our objective, two approaches have been investigated: *"Hardware Accelerators"* and *"Approximate Computing"*.

Towards achieving the objective of this dissertation, we contributed to the research community with several ideas that can be summarized as:

- An exact and approximate k-Nearest Neighbor (kNN) classifiers are proposed with a classification accuracy comparable with [5] for a touch modality recognition task. The experimental results demonstrate the effectiveness of Data-oriented ACTs on reducing the memory usage and execution time of the KNN classifier with an acceptable accuracy loss.

- An approach for applying algorithmic level ACTs on machine learning algorithms is proposed. Each ACT in the proposed approach can serve as a quality configurable knob to trade-off quality for time latency.

- An overview about energy efficient implementation of machine learning algorithms on hardware platforms highlighting the main challenges when embedding such algorithms is provided. Moreover, we report the techniques that could be applied to improve the energy efficiency. Furthermore, the main factors to be taken into consideration when choosing the appropriate platform are highlighted. Lastly, the strategies to overcome the challenges when building an energy efficient embedded machine learning systems are discussed.

- A book chapter that presents a survey of the existing algorithms and tasks applied for tactile data processing. The presented algorithms and tasks include machine learning, deep learning, feature extraction, and dimensionality reduction. Moreover, this chapter

provides guidelines for selecting appropriate hardware platforms for the algorithm's implementation. The algorithms are compared in terms of computational complexity and hardware implementation requirements.

- A comprehensive assessment of applying algorithmic level approximate computing techniques on the FPGA implementation of tensorial Support Vector Machine (SVM) has been performed.

- An architecture for Singular Value Decomposition (SVD) computation based on approximate computing techniques is proposed. The architecture is based on a shallow neural network for finding the SVD of an input matrix with two different dimensions. We provide the structure, tuning, and training of the network. Also, the FPGA implementation of the proposed neural network inference is presented. Implementation results show that the proposed network achieves a significant speedup and reductions in the required hardware resources and power consumption respectively compared to the traditional one-sided Jacobi algorithm.

- The first hardware implementation of a neural network based SVM featuring multidimensional tensorial inputs is proposed. The implementation is feasible for real-time touch modality classification with low power consumption. Moreover, the implementation scalability shows that the neural network based SVM is adequate for the acceleration of SVM on resource-limited hardware platforms.

- The design and implementation of a kNN accelerator using a selection based sorter (Selector) is proposed. The proposed accelerator overcomes similar state of the art solutions by reducing the occupied hardware area while providing noticeable speedups.

- A Hybrid fixed-point binary Convolution Neural Network (HCNN) model for touch modality classification is presented. A hardware accelerator architecture and implementation on FPGA for HCNN is proposed. The proposed accelerator can classify an input touch with a higher accuracy compared to SVM and Deep CNN. Moreover, a faster classification time is noticed while providing a low energy per classification value.

1.2 Dissertation Outline

1.2.1 State of the art

This chapter explores the existing learning algorithms used for tactile data processing. Starting from pre-processing to classification/regression algorithms. Then, the computational complexity of such algorithms is studied. Moreover, a discussion is presented on the existing hardware platforms that could be adequate for the acceleration of learning algorithms. Although not all machine learning algorithms have been used for embedded tactile processing, existing accelerators architecture can be studied and modified for most applications. Thus, in this chapter a survey on efficient hardware accelerator design is provided. The survey focuses in principle on kNN, SVM, and Binary CNNs. Similarly, existing approximate computing techniques targeting these algorithms is tackled in this chapter. The techniques mainly belong to the algorithmic and architectural levels.

1.2.2 Algorithmic Level Approximate Computing Techniques for Machine Learning

An approach for applying algorithmic level approximate computing techniques on machine learning algorithms is proposed. The approach is validated on the software and hardware implementations of kNN and Tensorial SVM (TSVM). The assessment of the software implementation shows that tactile data classification can be accelerated with a reduced memory usage while running kNN algorithm on an Intel CPU. As for hardware implementation, both the approximate kNN and TSVM are able to process tactile data in real-time with a noticeable power reductions compared to their exact counterpart when implemented on FPGA. The obtained efficient approximate implementations are achieved with a classification accuracy loss than 10%.

1.2.3 Efficient Selection-Based k-Nearest Neighbor Architecture on Modern SoCs

In this chapter, we propose a novel selection-based sorter (Selector) to be embedded in the the architecture and implementation of kNN algorithm. The selector idea is determine the k-nearest neighbors without sorting the complete distance vector. The selection architecture is configurable with a division factor to control the process based on the target application. Compared to existing traditional sorters, the selector offers significant speedup with a reduced hardware area at the best, and matches the Quick sorter performance in the worst

case. An Exact and Approximate selection-based kNN implementations are proposed. The approximate architecture utilizes the use of algorithmic level approximate computing techniques. When validated on a touch modality classification problem, both the proposed exact and approximate kNNs offer a real-time classification while consuming low energy when implemented on Xilinx Zyqnberry platform. Such efficient kNN implementations are achieved with an accuracy degradation of at most 2.6%.

1.2.4 Real-time Accelerated Tensorial Support Vector Machine Architecture

This chapter introduces an efficient architecture of the tensorial SVM algorithm based on the use of a Shallow Neural Network (NN) for the Singular Value Decomposition (SVD). A detailed process for the design, training and tuning of the proposed network is presented. Such network architecture is capable of predicting the singular vectors with significant reductions in the FPGA implementation footprint compared to the traditional one-sided Jacobi algorithm. To validate the effectiveness of the proposed solution, a scalability assessment of the proposed NN-TSVM architecture is performed. The assessment shows that replacing the one-sided Jacobi with a neural network demands only 1% increase in the required hardware area compared to 29% when the number of training tensors is doubled. Moreover, the efficient NN-TSVM implementation is capable of real-time tactile data classification with a reduced energy consumption that is suitable for the implementation on resource-limited platforms such as the Zynqberry.

1.2.5 A Hybrid Precision Architecture for an Efficient Binary Convolutional Neural Network Accelerator

In this chapter, a hybrid fixed-point/Binary convolution neural network accelerator is proposed as a trade-off between the reliability of CNNs and the low complexity of Binary CNNs. The architecture adopts a complete binarization of the hidden layers and 16-bit fixed-point for the first and last layers activations with binary weights. A design methodology is provided in terms of network topology, placement of binarization layers, and training process. The proposed Hybrid Binary Weight Network (H-BWN) achieves an accuracy increase up to 35% for the classification of touch modalities compared to traditional BCNN topology. The H-BWN accelerator offers real-time classification with an energy per classification that represents a fraction of the energy consumed by existing similar solution targeting FPGA implementations. Such results pave the way towards the deployment of the intelligent tactile

data processing on small mainstream microcontrollers with a storage requirements of less than 5 KB.

1.3 List of Publications

This doctoral research has resulted in several publications, a list of which is provided here:

Journals:

1. H. Younes, A. Ibrahim, M. Rizk, and M. Valle, "A Shallow Neural Network for Real-Time Embedded Machine Learning for Tensorial Tactile Data Processing," IEEE Transactions on Circuits and Systems I: Regular Paper, vol. 68, no. 10, pp. 4232–4244, Oct. 2021.

2. H. Younes, A. Ibrahim, M. Rizk, and M. Valle, "An Efficient Selection-Based kNN Architecture for Smart Embedded Hardware Accelerators," IEEE Open Journal on Circuits and Systems,vol. 2, pp. 534–545, Aug. 2021

3. H. Younes, A. Ibrahim, M. Rizk, and M. Valle, "Algorithmic-Level Approximate Tensorial SVM Using High-Level Synthesis on FPGA," Electronics, vol. 10, no. 2, p. 205, Jan. 2021.

4. A. Ibrahim, H. Younes, M. Alameh, and M. Valle, "Near Sensors Computation based on Embedded Machine Learning for Electronic Skin," Procedia Manufacturing., vol. 52, pp. 295–300, 2020.

Conference Papers:

1. H. Younes, A. Ibrahim, M. Rizk, and M. Valle, "Efficient FPGA Implementation of Approximate Singular Value Decomposition based on Shallow Neural Networks," in 2021 IEEE 3^{rd} International Conference on Artificial Intelligence Circuits and Systems (AICAS), Washington DC, USA, Jun. 2021, pp. 1– 4.

2. H. Younes, A. Ibrahim, M. Rizk, and M. Valle, "Algorithmic Level Approximate Computing for Machine Learning Classifiers," in 2019 26^{th} IEEE International Conference on Electronics, Circuits and Systems (ICECS), Genoa, Italy, Nov. 2019, pp. 113–114.

3. M. Osta, M. Alameh, H. Younes, A. Ibrahim, and M. Valle, "Energy Efficient Implementation of Machine Learning Algorithms on Hardware Platforms," in 2019 26^{th} IEEE International Conference on Electronics, Circuits and Systems (ICECS), Genova, Italy, Nov. 2019.

4. H. Younes, A. Ibrahim, M. Rizk, and M. Valle, "Data Oriented Approximate k-Nearest Neighbor Classifier for Touch Modality Recognition," in 2019 15^{th} Conference on Ph.D Research in Microelectronics and Electronics (PRIME), Lausanne, Switzerland, Jul. 2019, pp. 241–244.

5. H. Younes, A. Ibrahim, M. Rizk, and M. Valle, "A Mixed Precision Binary Neural Network Architecture for Touch Modality Classification" in 16^{th} Conference on PhD Research in Microelectronics and Electronics (PRIME), Germany, Jul. 2021.

6. H. Younes, A. Ibrahim, M. Rizk, and M. Valle, "Hybrid Fixed-point/Binary Convolutional Neural Network Accelerator for Real-time Tactile Processing," in the 28^{th} IEEE International Conference on Electronics Circuits and Systems (ICECS), Dubai, UAE, 2021.

Book Chapters:

1. H. Younes, A. Ibrahim, M. Alameh, and M. Valle, "Efficient Algorithms for Embedded Tactile Data Processing," in Electronic Skin: Sensors and Systems, River Publishers Series in Electronic Materials and Devices, 2020. [Online]. Available: https://www.riverpublishers.com/book_details.php?book_id=789.

Chapter 2

State of the art

The idea behind hardware accelerators is to determine complex and demanding blocks of an algorithm, then assign a high performance module (namely referred to as Processing Element (PE)) to execute such block in an efficient manner. Designing a hardware accelerator can be performed using several methodologies. The use of "specific design" methodology suitable for the hardware platform and application. For example, modern System-on-Chips (SoCs) include an ARM CPU and a Zynq FPGA on the same board (e.g. ZedBoard, Zynqberry). Such design allows the execution of complex and parallel portions of an algorithm on the FPGA while offloading simple tasks to the CPU. Another designing methodology is the use of high performance cores in multi-core CPUs/GPUs for demanding tasks and assign less performing cores for simpler tasks. This can be found in Apple new designed M1 chips that includes both "High Performance Cores" and "Efficient Low Power Cores" [6]. Recently, a new hardware accelerator design methodology focuses on techniques such as In-Memory Computing (IMC), Near-Memory Computing (NMC), and Processing-In-Memory (PIM), have emerged to bring computing as close as possible to the memory array such as, allow to reduce the cost of data movement between computing cores and memories [7].

In this dissertation, we are focused on the "specific design" methodology targeting different machine learning algorithms and hardware devices/platforms. Mainly, the design and implementation of hardware accelerators for kNN, SVM, and Binary CNNs. Then, a set of approximate computing methods for enhancing the accelerators' performance is discussed. The performance of the exact and approximate accelerators is assessed on a tactile data processing application, mainly touch modality classification.

2.1 Tactile Data Processing

Tactile sensing involves the detection of motion, the measurement of contact parameters, the processing of the signals to extract structured and meaningful information, and the transmission of such information into a higher system levels for interpretation [8]. The data acquired from tactile sensors corresponds to an electrical stimulus. The latter varies according to the type of the sensing material, dimensionality, responsiveness, and structure of the sensor. Processing algorithms should be able to decode and efficiently handle the acquired data. Tactile data processing algorithms presented in the literature could be divided into two categories: pre-processing and classification/regression. Pre-processing algorithms involve feature extraction and dimensionality reduction, while classification and regression algorithms are mainly machine learning algorithms.

2.1.1 Pre-Processing

Tactile data may be pre-processed to reduce noise and extract meaningful features. The extracted features could be (1) the variables that best describe the raw data and (2) the weights which should be given for each variable. For instance, sub-sampling can be applied to a recorded touch reading to remove silent/noisy samples. Also, data obtained from certain taxels in the sensor patch can be considered in a pattern recognition problem. These taxels are the ones that provide reliable data (nonzero or unknown readings). Several algorithms have been presented in the literature for dimensionality reduction and feature extraction such as Principal Component Analysis (PCA), Independent Component Analysis (ICA), and Linear Discriminant Analysis (LDA).

Principal component analysis is the base for multivariate data analysis (i.e. studying the effect of multiple variables on the output state) [9]. PCA is used for approximating data or reducing the dimensionality of the data e.g. representing data from X_n space in X_{n-k} space, where n and k are two positive integers. As a concrete example, if we have data with n features, then PCA helps to represent these data with $n-k$ features with the least possible losses. Figure 2.1 shows how PCA can be applied to reduce dimensionality from three dimensions (3D) to 2D (the figure has been generated using the data and code provided in [10]). In [11], a finger-like shape tactile sensor has been used to collect data about fabric surfaces. Initially, Fast Fourier Transformation (FFT) has been used to construct the original dataset, and then PCA has been applied to compress the attribute data and extract feature information. In [12], kernel PCA [13] has been used for low-resolution tactile image recognition for automated robotic assembly. Kernel PCA is a method to perform a nonlinear form of the PCA. It computes higher-order statistics among random variables while reducing

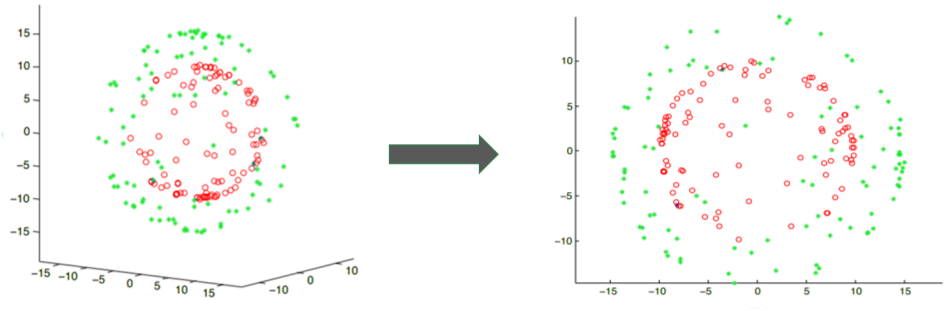


Fig. 2.1 PCA Example: 3D to 2D

the data dimensionality, thus being able to achieve the goal of both feature extraction and dimensionality reduction. The authors in [14] have used local PCA [15] combined with a neural network to classify 16 household and toy objects. Local PCA is a nonlinear extension of the normal PCA. It has been used to obtain a less complex feature vector for the data obtained from tactile sensors mounted into a robotic arm.

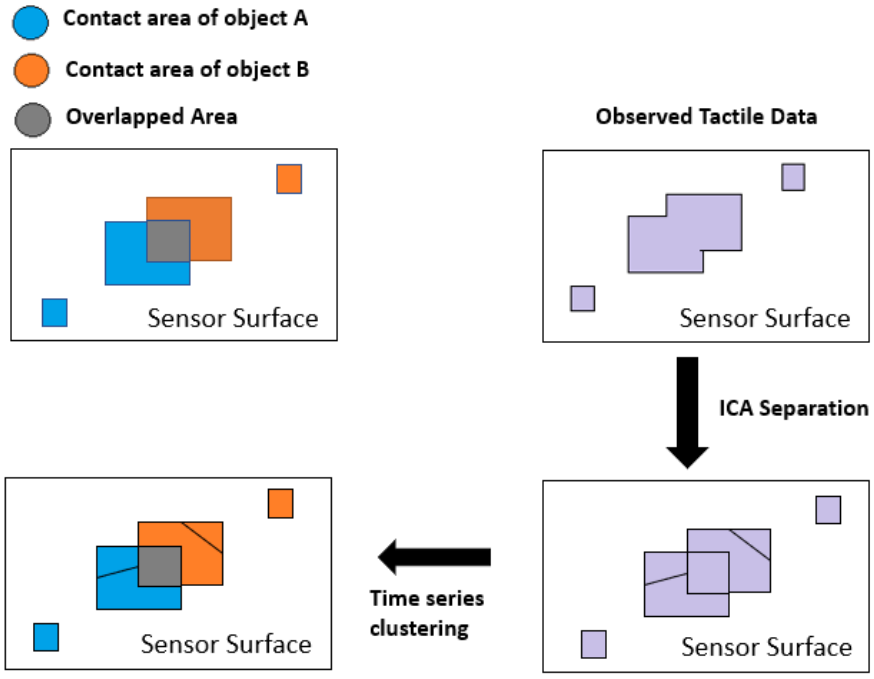


Fig. 2.2 Procedure of tactile data separation using ICA

Independent component analysis [16] can be seen as an extension of the PCA. It is a linear dimensionality reduction technique, which searches for the linear transformation that reduces or eliminates the linear dependency between elements of a random vector. An example of using ICA is the Cocktail Party Problem [17]. Spatial ICA has been adopted as a separation

method that allows a robot to understand and interact with tactile information from multiple sources [18]. Figure 2.2 shows the procedure of tactile data separation from two objects using ICA along with time series clustering.

Linear discrimination analysis shown in Figure 2.3 is yet another method for dimensionality reduction. It consists of finding the projection hyperplane that minimizes the variance within the same class, and maximizes the distance within the projected means of the classes [19]. Tactile images of deformable and non deformable surfaces have been used for a classification problem in [20]. LDA has been used as a separation algorithm between six different surfaces with an accuracy rate of up to 95.5%. In [21], the authors have demonstrated the feasibility of using LDA for surface texture discrimination. Another use of LDA appears in [22] for terrain discrimination problems.

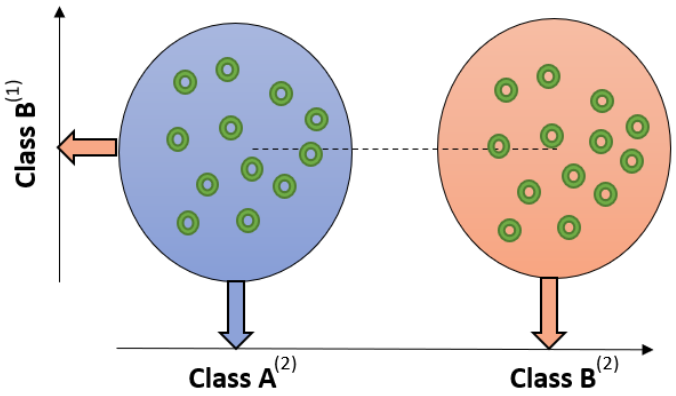


Fig. 2.3 Linear Discrimination Analysis: (1) Bad projection and (2) Good Projection

2.1.2 Classification and Regression

Machine learning algorithms are an efficient solution for processing tactile data in various applications [5]. ML algorithms in general, can extract a complex, non linear input–output relationship based on learning by example approach. An ML algorithm is trained using a set of examples, where each example is described by a group of informative features. ML algorithms can support intelligent and predictive systems that can make accurate decisions on unseen data. In this perspective, classification/regression problems supported by ML algorithms could be adopted in applications with tactile information including:

- Normal force sensing for e.g. grasp control, object manipulation, touch modalities, and orientation determination;

Table 2.1 Learning Algorithms for Tactile Data Processing Applications

Tactile Application	Learning Algorithm	Examples
Object Manipulation	Decision Tree, Naive Bayes, Support Vector Machine, k-Nearest Neighbor, Convolution Neural Networks, Long Short-Term Memory	[23], [24], [25], [26]
Surface Texture Identification	Least Square Support Vector Machine, Expectation-Maximization, Regularized Least Square, Bayesian Exploration, Reinforcement Learning	[27], [28], [29], [30]
Touch Modality Classification	k-Nearest Neighbor, Decision Tree, Logitboost, Extreme Learning Machine, Support Vector Machine, Regularized Least Square, Deep Convolution Neural Networks, Long Short-Term Memory	[5], [31], [32], [29]
Slip and Grasp Detection	K-Means Clustering, k-Nearest Neighbor, Support Vector Machine, Random Forest, Locally Weighted Projection Regression	[33], [34], [35], [36]
Orientation Determination	Convolution Neural Network, Linear Regression, Spiking Neural Network	[37], [38], [39]

- Shear force sensing for e.g. grasp control and friction determination;
- Vibration detection for e.g. slip detection and texture determination.

Table 2.1 reports the commonly used ML algorithms with respect to the type of extracted tactile data. Object compliance classification problem involves the identification of objects' structure (soft, hard, fuzzy, bumpy, etc.) and state (ideal, moving, etc.). For a vegetable grading industry, differentiation between green, moderate, and ripe tomatoes has been achieved with an accuracy of 90% and 85% using Decision Tree (DT) and Naïve Bayes (NB) respectively [23]. A classification problem involving 18 objects of different size, compliance, and state (fixed, moving) has been tackled in [25]. Using a k-Nearest Neighbor (kNN) classifier with $k = 2$, an object's structure classification accuracy up to 80% was attained. While, for detecting the state of the object, an accuracy of 91.4% was obtained for the given problem. In [26], a Convolutional Neural Network (CNN) has been trained on 5300 instances collected from 53 objects with different compliance. A classification accuracy up to 85% has been reached on identifying the compressibility and smoothness of unseen objects.

Recognition and categorization of object properties can be obtained by analyzing the surface texture. Using Support Vector Machine (SVM) algorithm, an autonomous humanoid robot recorded a recognition rate up to 100%, with 70% objects' categorization ability in a setup that involved ten different objects (glass, sponge, paper, etc.) [27]. In [28], Least Square SVM was adopted to discriminate 20 daily used objects based on their texture. A classification accuracy between 70% and 100% has been recorded when using 10 training samples. Bayesian Exploration and Reinforcement Learning have been used to train and validate a discrimination system in [30]. The system was able to differentiate between 10 objects (brick, copper, wood, etc.) with a 90% success rate.

Slip and grasp detection is an another task that can be supported by learning algorithms. In [33], a Phantom Omni arm has been equipped with a tactile sensory array of 84 sensor cells to study the translational and rotational movement of an object. The arm was able to hold

and recognize a ball with an accuracy of 91.2% using K-Means clustering algorithm. The humanoid robot ARMAR-IIIB was learned to grasp objects using SVM algorithm [34]. The grasp was recorded as successful or not by the ability to lift up the object. 77% of the grasps were considered as stable compared to 23% unstable tries. In [36], a Haptically-enabled robot with Barret arm system which three fingers are equipped with BioTac sensors was used to achieve a reliable grasping of fragile objects. The best performance was achieved using a 3-layer neural network regression by detecting a slip within 30 ms with 80% success rate.

Touch Modality classification allows the integration of gesture-based actions that can be performed by robots or humans with prosthetic hands. For a medical purpose of caring for patients with mild mental impairment; a humanoid equipped with artificial skin has been trained to discriminate between nine touch modalities (scratch, tickle, rub, etc.) [31]. Recognition rates up to 96.7% has been achieved using four machine learning algorithms including kNN, SVM, DT, and Logitboost. The authors in [29] have adopted a touch modality classification problem that involves three patterns: brushing a paint brush, rolling a washer, and sliding a finger on 4×4 tactile sensory array. SVM and Extreme Learning Machine (ELM), and Deep CNN based on transfer learning have been chosen as learning algorithms. A classification accuracy of 90%, 89.6%, and 76.9% has been recorded respectively.

2.2 Embedded Machine Learning

Embedding machine learning in resource-limited and battery-powered applications for tactile data processing must obey a set of requirements including: small hardware area, low time latency, and low energy consumption. The main two factor that affects such requirements are the computational complexity of these algorithms and the hardware device/platform.

2.2.1 Computational Complexity of Machine Learning Algorithms

Table 2.2 provides the Big-O complexity of the most used algorithms for tactile processing [40], where n is the size of the training set, f is the number of features, n_{trees} is the number of trees, and n_{SV} is the number of support vectors. The complexity of CNN is based on the information provided in [41]. Only convolutional layers are considered as it is assumed that fully connected layers constitute only 5%-10% of the CNN complexity. Here l is the index of a convolutional layer, d is the number of convolutional layers, n_l is the number of filters in the $l-th$ layer, n_{l-1} is the number of input channels of the $l-th$ layer, s_l is the spatial size of the filter, and m_l is the spatial size of the output feature map.

Table 2.2 Computational complexity of machine learning algorithms

Algorithm	**Application**	**Training**	**Testing**
Naive Bayes	Classification	$O(nf)$	$O(f)$
Decision Tree	Classification/Regression	$O(n^2 f)$	$O(f)$
SVM	Classification/Regression	$O(n^2 f)+n^3$	$O(n_{SV} f)$
kNN	Classification/Regression	-	$O(nf)$
Linear Regression	Regression	$O(f^2 n+f^3)$	$O(f)$
Random Forest	Classification/Regression	$O(n^2 f n_{tress})$	$O(f n_{tress})$
CNN	Classification/Regression	$O(\sum_{l=1}^{d} n_{l-1}.s_l^2.n_l.m_l^2)$	

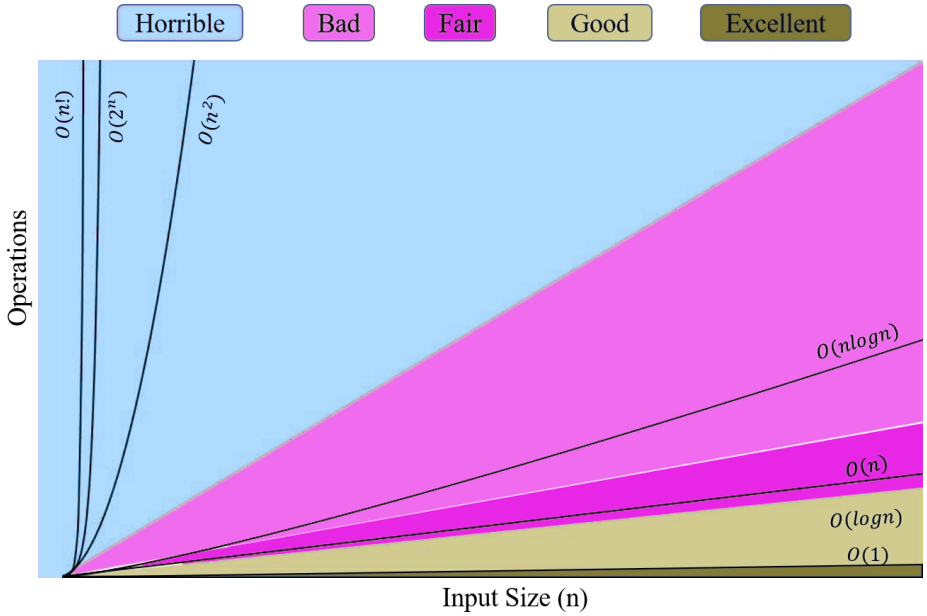


Fig. 2.4 Big-O Complexity

The complexity given in Table 2.2 has been analyzed based on the degree of complexity provided by Figure 2.4. It is noticed that algorithms such as DT and SVM involve complex training phase that increases quadratically for a large number of training points. For linear regression (LR), the training phase complexity also increases quadratically with the number of features, which is usually less than the number of training points. Meanwhile, the DT, SVM, and LR classification phases are relatively less complex. For NB, the training phase is less complex compared to the SVM and DT with a low complexity classification phase too. This is due to the linear complexity compared to the quadratic one in the case of the SVM and DT. Similarly, the complexity of the classification phase of the kNN increases linearly with the increase in the number of training points and the number of features. Since

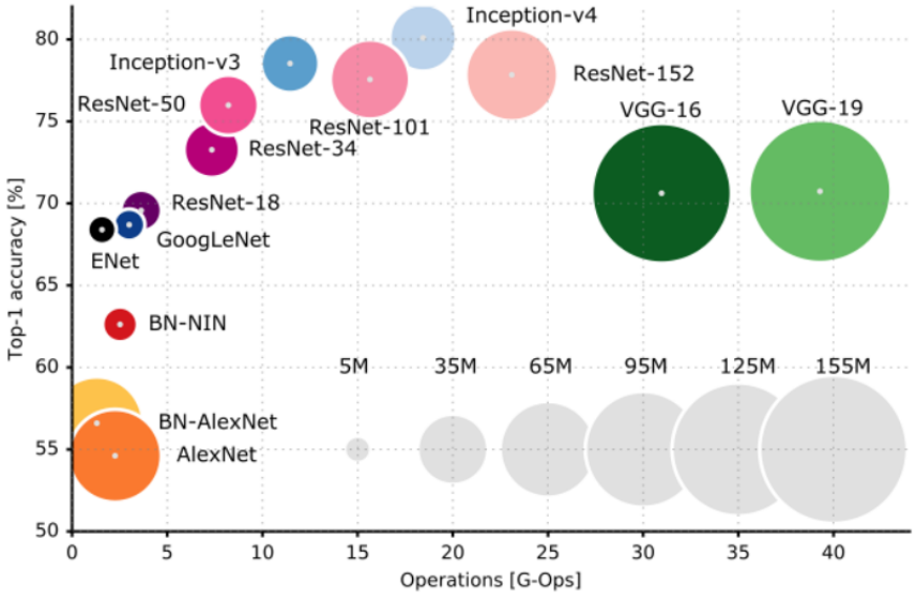


Fig. 2.5 Top-1 one-crop accuracy versus amount of operations required for a single forward pass. The size of the blobs is proportional to the number of network parameters

kNN does not have a separate training phase, hence to classify a new input the distance with respect to all training points is to be calculated each time, its usage is efficient mostly for small training set size. As for the complexity of CNNs, it depends on the configuration of the network compared to other algorithms, which is affected more by the filter and output feature maps size compared to the number of filters and input size. Figure 2.5 shows the the complexity of the existing CNNs in terms of the number of operations and their classification accuracy [42]. These networks differ in the number of layers, size of filters, number of filters, etc. which verifies the the presented complexity in Table 2.2.

2.2.2 Hardware Platforms

The hardware platform must be able to handle the complexity of the algorithm while achieving the expected performance in terms of area, time latency and energy consumption. A wide selection of hardware devices and platforms maybe used to implement the tactile processing algorithms. Some sound and widely used devices include Field Programmable Gate Array (FPGA), Graphics Processing Unit (GPU), Microcontroller Unit (MCU), Parallel Ultra Low-power Platform (PULP), Tensor Processing Unit (TPU), Application-Specific Integrated Circuit (ASIC), and platforms include Raspberry Pie, ZedBoard, Zynqberry, Python Productivity for Zynq (PYNQ), etc. The available devices and platforms differ in size, target programming language, area utilization (LUT, FF, DSP, BRAM, etc.), maximum operating

Table 2.3 Comparison between different hardware devices/platforms

Device/Platform	Target Implementation	Framework/ Programming Language	Strengths	Weaknesses
FPGA	Hardware	VHDL, Verilog, C/C++ with OpenCL, SDAccel, HLS	High performance per watt, parallelism	Not suited for floating-point operations, long development time, programming difficulty
GPU	Software	OpenCL, NVIDIA CUDA, C/C++, Java, Python	Massive processing power for image, video and signal processing	High power consumption, need for API frameworks to take advantage of parallelism
ASIC	Hardware	Application-specific, ex: TensorFlow for TPUs, tools from manufactures	Optimum combination of performance and power consumption	High cost, longest development time, Unconfigurable
PULP	Software	C language only	Low power consumption, tunable performance, Open source	Low size on-chip memory, Development Difficulty
ZedBoard/Zynqberry	Hardware/Software	Adopts characteristics of FPGA and ARM processors Pluses: Ability to use FPGA as hardware accelerator, Linux Development		
PYNQ	Hardware/Software	Adopts characteristics of FPGA and ARM processors Pluses: Python Programming, Arduino and Raspberry Pie shield connectors		

frequency, etc. Table 2.3 presents the common characteristics of the most used devices and platforms related to the variety of machine and deep learning applications. These devices could be utilized for either hardware or software development applications. For instance, FPGAs and ASICs are used for hardware implementations, while MCUs and GPUs are intended for software implementation. As for hybrid boards such as Zynqberry/PYNQ, they can be used for both hardware and software implementations as they include both an FPGA and an ARM CPU.

In principle, the implementation of machine learning training and testing phases are not correlated. Hence, different devices can be assigned. For example, a complex neural network can be trained using a GPU due to the available high memory bandwidth and a large number of processing cores. Then, the inference can be accelerated using an FPGA or an MCU depending on the network size. From application perspective, a large sized GPU/FPGA is not adequate for space limited scenarios (e.g a smart watch). The same can be said for implementing neural network training on an MCU. Besides, the development time is a key issue to consider, especially for complex machine learning algorithms such as tensorial SVM using VHDL language. For a finalized implementation which is later to be commercialized, an ASIC can be considered as the best available option as no updates are scheduled until a new generation release. Another aspect for choosing a hardware platform is the cost. For example, if the cost difference between a GPU and CPU is not proportional to the gain in speedups, it is not worth paying the extra cost of the relatively expensive GPUs.

2.2.3 Machine Learning Accelerators

Knowing that not all machine learning algorithms have been adopted for embedded tactile data processing, nonetheless accelerators are discussed in terms of design methodology, architecture, and acceleration gain in terms of hardware implementation footprint. Then,

through the following chapters, novel hardware accelerators for embedded tactile data processing are proposed based on the techniques reviewed in this chapter.

2.2.3.1 k-Nearest Neighbor Accelerators

The complexity of the kNN algorithm lies in the large number of distance calculations towards all the samples in the training set. Due to the fact that both the size and the dimensionality of datasets have been rapidly growing. Thus, kNN can be mainly accelerated by reducing the memory footprint due to the training set and the distance calculation operation itself. For this purpose, several techniques have been proposed to accelerate kNN implementations on FPGA with different end targets. Table 2.4 summarizes the key techniques used with emphasis on their aim and the acceleration gain obtained as follows:

Reduce Memory Accesses: To reduce the impact of memory access constraint imposed by the extensive distance calculation operations, several methods have been proposed. Starting with reducing the data representation from floating-point to fixed-point with low precision. This method decreases both the memory storage of the data and the corresponding arithmetic operations. Such decrease can be also achieved with the use of PCA [43]. PCA aims at reducing the dimensionality of the data, thus simplifying the distance calculation operation. Another method is to dispatch data from external memory (e.g. DRAM) through a Direct Memory Access (DMA) engine in bursts to reduce the memory access overhead [44].

Scalability and Re-configurable Architectures: Although the performance of kNN is affected by the number of training samples (N) and number of neighbors (k), these parameters have a direct impact on the scalability of a kNN implementation. One of the most used techniques to design a scalable accelerator is the adoption of a systolic array of processing elements [45]. Each processing element is responsible for a specific operation (e.g. distance calculation, sorting, etc.). To obtain a scalable architecture, each systolic array is designed with different value of (k), assigned for a different core (for multi-core architectures), and can operate on different input size either in serial or parallel based on the scalability level required by the application. The same methodology can be used to design a re-configurable kNN architecture targeting multiple FPGAs connected through a set of memory controllers on a co-processor (e.g. Convey HC-2 [46]).

Parallel Implementations: The distance calculation between two samples of M features can be performed in parallel since the features are uncorrelated. Similarly, the distance calculation between samples A and B is independent of the distance calculation between samples A and C. Hence, distance calculation operations can be fully or partially implemented in parallel. This can be easily designed with the use of OpenCL framework or HLS directives.

Table 2.4 kNN Hardware Accelerators: Design Target, Techniques and Noticeable Gains

Target	Key Techniques	Acceleration Gain	Reference
Reduce Memory Accesses	Low precision data representation	DRAM data access reduced by up to 231.4× using 4-bit data precision	[43]
	Principle Component Analysis based Filtering (PCAF)	FPGA Performance equivalent to that of a 56-thread CPU server	
Scalability	Separate PE design and optimization for each kNN block	Up to 76× speedup compared to GPP	[45]
	Systolic array of PE	Resources utilization proportional to k	
Optimizing Parallel Implementation	Use of OpenCL parallel computing framework	Classifies about 290 objects/s	[47]
	Odd-Even sorter	An Energy per classification of 83 mJ	
Complete Deployment on FPGA	Use of High Level Synthesis (HLS) Optimization Directives	Speedups between 147× and 388× for k = (3, 5,7, and 10)	[48]
	Fixed-point representation		
Design of Soft Parameterized IP Cores	Block pipelining and parallel processing	Real-time classification of large datasets and number of features on medium-sized FPGAs	[49]
	Linear space-time mapping methodology	Area (CLB coverage) and performance scale with O(N)	
Acceleration on Resource-Limited Devices	Feature Extraction: Root Mean Square (RMS)	1% memory requirements on the Cyclone V FPGA	[50]
	Pipelined Distance Calculation		
Reconfigurable Architecture	Sparse Matrix Vector Multiplication	Speedup up to 1.5× compared to a 32-thread CPU	[46]
	Systolic array of PE	FPGA averages 2.72 billion term document elements per second	
Modular and Scalable Hardware/Software Architecture	Divide kNN workload between FPGA and CPU	Speedups as high as 67.4× and energy reductions up to 50× compared to ARM Cortex-A9 CPU	[44]
	Simplified Insertion Sorter	Varying k only affects the number of LUTs for FPGA implementation	
		Up to 24 kNN accelerators can run on a ZedBoard platform	
Classification of Large Scale Datasets	Predetermined Range Search (PRS)	Speedups up to 1.4× compared to similar designs	[51]
	Pipelined Architecture	Reduced BRAM utilization of 33%	

Another aspect that can be designed in parallel is the sorting process. An even-odd sorter has been proposed in [52] to make full use of the parallel pipeline structure of the FPGA.

Design of Soft Parameterized IP Cores: The need for real-time classification of data with large size and high dimensionality requires a kNN implementation of very high performance. This can be achieved through the design of parameterized kNN architectures that can be synthesized effortlessly for any desirable combination of parameters (N, M, and k) [49]. Using a space-time mapping methodology, kNN computation nodes can be modeled so that each node can perform one operation per cycle. Hence, each block of the kNN architecture can be assigned to a different set of parallel nodes customized for specific parameters.

Hardware/Software Architecture: The implementation footprint of the distance calculation is relatively higher than that of the sorting process. Hence, several hardware/software architectures have been proposed to benefit from such complexity difference. The idea is to assign a high performance hardware such as the FPGA for distance calculation, and offload the sorting process to a General Purpose Processor (CPU) [44]. This design methodology has been made possible with hybrid platforms (e.g. Zynqberry, ZedBoard, etc.) that has a Zynq FPGA and an ARM processor. Such design also paves the way towards running multiple accelerators on the same platform achieving further acceleration.

Application and Platform Specific Architectures: A quite few kNN accelerator architectures exists targeting a certain application, a certain hardware platform, or a certain set of requirements (e.g. power, time latency). In [48], a design exploration process is presented in order to identify the effect of the value of (k), and feature extraction on the implementation resource targeting large datasets. Each design has been optimized with HLS directives so that if fits completely on Zynq-700 FPGA. To support EMG recognition for gestures communication with limited hardware resources, authors in [50] proposed a feature extraction process to transform high dimensional data into a single dimension. Such process allowed the complete pipelining of the distance calculation resulting in accelerated kNN classification process. Targeting large datasets, a kNN accelerator based on Predetermined Range Search (PRS) method is proposed in [51]. The proposed architecture is characterized by a simple circuit structure achieving a significant decrease in the number of BRAMs.

2.2.3.2 Support Vector Machine Hardware Accelerators

In contrast to kNN, SVM has two separate training and classification phases. Hence, existing research work aims to implement and accelerate an SVM model during both phases. In this section we present a survey on SVM classification accelerators only. For reference, authors in [53] have investigated the acceleration of SVM training on FPGA. Various techniques have been developed for the hardware acceleration of the online classification process. The

Table 2.5 SVM Hardware Accelerators Targeting Different Kernels

Implementation Technique	Kernel Function	Acceleration Gain	Reference
Systolic Array Architecture	Polynomial	5% Power Reduction	[57]
	Linear	85× speed-up over equivalent GPP	[58]
Dynamic Partially Reconfigurable		8x reduction in reconfiguration time over the single non-DPR classifier implementation	[59]
Block-based Partial Storage		A throughput rate of 216 and 70 f/s for XGA (1024×768) and HD (1920×1080) video resolutions, respectively	[60]
Multiplier Less Kernel	Hardware Friendly	1.6 µs for one kernel calculation	[61]
		5% Utilization on Xilinx Spartan 3E	[62]
Pipelined Architecture		2 orders of magnitude in the average instance classification time, in comparison with software implementations	[63]
		Processes up to 33.8 640×480 image fps	[64]
		8-bit implementation with equivalent performance to floating-point	[65]
Design Specific (SPW, System Generator)	Linear/Gaussian/RBF	11% Utilization on Virtex-4	[66]
		2.53× faster classification than the software implementation	[67]
Custom Arithmetic	Gaussian/Polynomial /Sigmoid	3×-7× speedup compared to the CPU and other FPGA/GPU implementations	[68]
Custom Processor	RBF	Real-time pixel classification in 0.1 ms	[69]
Hybrid Processing Architecture	Linear/Polynomial	Speed-up of 5× over the single parallel SVM classifier Utilization of 43% fewer hardware resources and a 20% reduction in power	[70]
Memory Management and Multi-core Architecture	Tensorial	3.6× faster than an ARM Cortex M4F (STM32F40) Achieves 15× better energy efficiency	[56]
Data Organization		Reduces the number of operations per inference from 545 M-ops to 18 M-ops and the memory storage from 52.2 KB to 1.7 KB	[71]
Hardware specific Optimization		9× faster than an ARM Cortex M4 at 168MHz with the same power consumption	[72]
Cascaded and Parallel CORDIC		Real-time touch classification with a peak of 302 G-ops	[73]

SVM models have been trained offline on software, then the models' parameters (e.g. support vectors, kernel factor, etc.) have been extracted to be used for classification. Table 2.5 presents the different developed techniques targeting different kernel functions. Tactile data is often represented in the form of a tensor as it preserves the structure of the raw data acquired from tactile senors [5], [54], [55]. Hence, the techniques used for accelerating the SVM algorithm targeting a tensorial kernel can be summarized as follows:

Memory Management and Multi-core Architecture: One of the essential operations in SVM targeting tensor input data is the unfolding process. Such process outputs three matrices that are required for further computations. Then, symmetrization and Singular Value Decomposition (SVD) are applied on each matrix. Thus, the whole process requires a lot of memory storage and data exchange, which is a challenging task on resource-limited devices. Authors in [56] have presented a memory allocation sequence that can be applied to the tensorial SVM on Parallel Ultra Low Power (PULP) Mr. Wolf microprocessor as shown in Figure 2.6. The allocation idea is based on 2D data transfers with a set of multiple banks where data allocation/de-allocation process can be serialized efficiently. Such technique has been validated by implementing the tensorial SVM by exploiting the multi-core architecture of Mr. Wolf. Results showed that a speedup up to 3× can be achieved compared to ARM Cortex-M4 by adopting the memory allocation sequence and using 2-cores for SVD calculation .

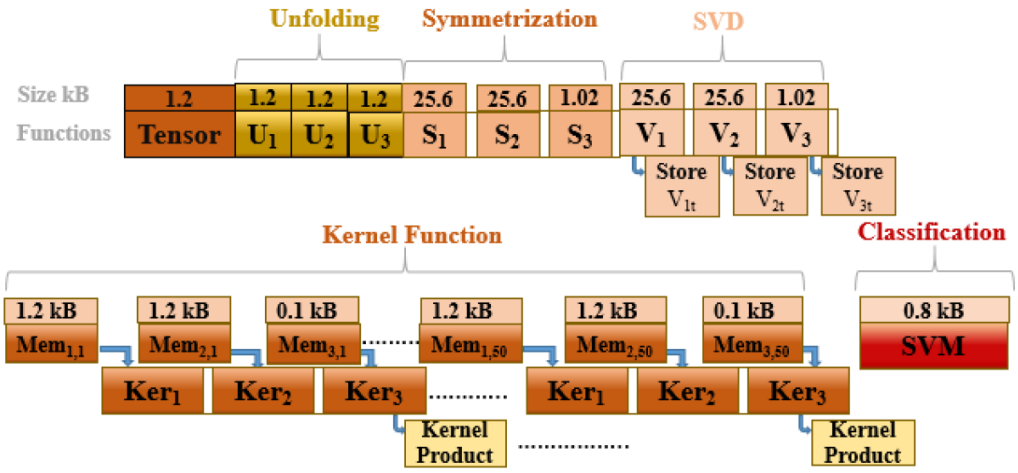


Fig. 2.6 Memory Allocation Sequence

Data Organization: For online and real-time classification, acquiring tactile data directly from sensors should be organized in local buffers to be processed. Authors in [74] have presented a scheduling algorithm to maximize CPU utilization with different L1 to/from L2 data transfers in the PULP. This scheduling allows the data from sensors to be continuously transferred to a circular buffer in L2 memory bank as shown in Figure 2.7. Each buffer contains a sample of 1 second. The platform is capable of performing 1.3 iterations at 20 samples per second. Such results presents a 9× speedup compared to that of ARM Cortex M4 with the same power consumption.

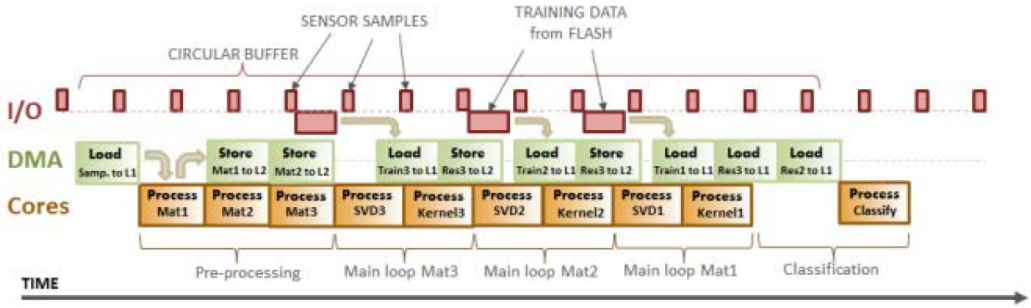


Fig. 2.7 Scheduling inside PULP

Cascaded and Parallel CORDIC: SVD is the most demanding and complex operation within the tensorial SVM algorithm [75]. The majority of the research presented in the literature adopts the One-sided Jacobi algorithm for SVD implementation whose main core is the Coordinate Rotational Digital Computer (CORDIC) algorithm. The first hardware acceleration of SVM featuring multidimensional tensorial inputs has been presented in [73]. The accelerator design offers two different architectures based on cascaded and parallel SVD

computation using the CORDIC algorithm. The cascaded implementation offers low power consumption but fails to offer real-time touch classification. The parallel implementation solved this problem and achieved a peak performance of 302 G-ops while consuming 1.14W for the Virtex-7 FGPA.

2.2.3.3 Binary Convolution Neural Network Hardware Accelerators

Binary convolution neural networks are characterized by fast computation, low power consumption and low memory footprint, which facilitates their deployment on different hardware such FPGA, ASIC, CPU, etc with limited computational resources and power budget. The XNOR-Bitcount operations in BCNNs allow for customizing data paths and optimizing the design making FPGAs as the most widely used platforms for BCNN deployment [76].

Quantization and network topology have been widely investigated for the design of BCNN hardware accelerators on FPGA. Figure 2.8 summarizes a survey (based on the data provided in [76] from more than 100 references and recent related work in [77], [78], [79], [80]) about the bit-width(**w/a**) that has been adopted for a BCCN architecture targeting commonly used datasets such as CIFAR-10 [81], ImageNet [82], etc.

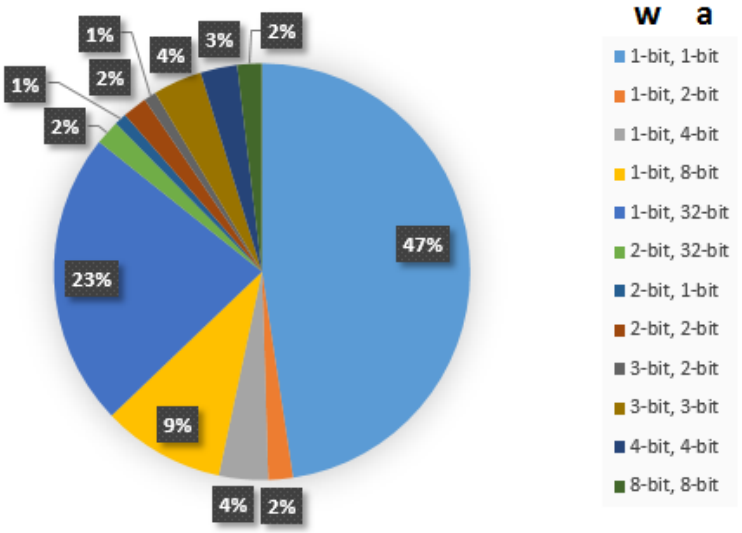


Fig. 2.8 Distribution of the most used Quantization Level

It can be seen that a complete binarization of CNNs is widely used for hardware acceleration (e.g XNOR-Net in [83] and DoReFa-Net in [84]). However, around 23% of the existing BCNN accelerators have adopted a 1-bit and 32-bit quantization for the weights and activations respectively (e.g BinaryConnect in [85] and ABC-Net in [86]). This leads to a claim that the binarization of activations in a CNN produces an accuracy loss more

than the binarization of weights. Other hardware accelerators have reported the use of n-bits ($n << 32$) quantization (e.g RetinaNet with $n = 8$ in [87]) as a compensation for the complexity increase imposed by keeping the activations in 32-bit representation.

In [88], authors have experimented with the effect of quantization/binarization on the weights and activations on the first and last layer of a small CNN trained on CIFAR-10 dataset as shown in Figure 2.9. The results backup the claim regarding that the binarization of network activations leads to a significant accuracy drop compared to the binarization of the weights. Moreover, for the use-case presented in [88], binarization of the weights of a CNN offers a similar classification accuracy compared to a 32-bit floating-point trained network.

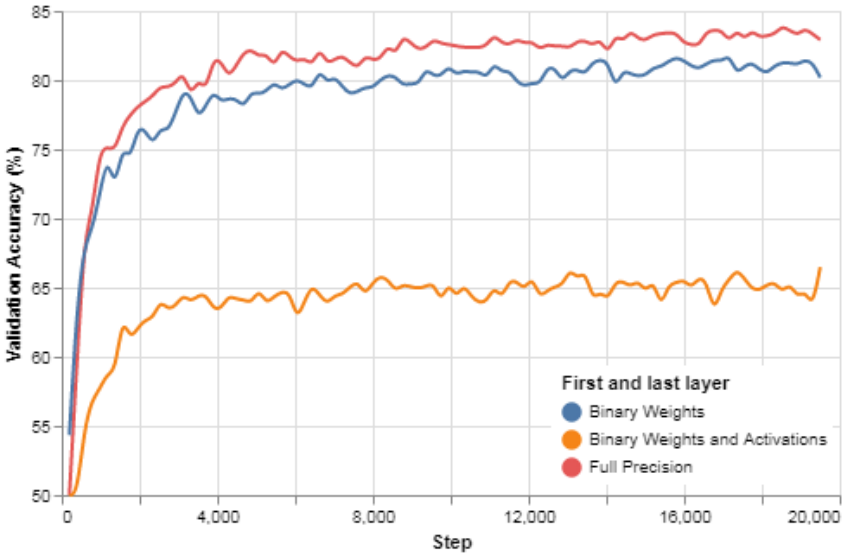


Fig. 2.9 A small CNN that has been trained on CIFAR-10 dataset with different precision for the first and last layers

Regarding the network topology considered when designing a BCNN accelerator, we extracted the reported topology in the survey used for the quantization distribution in Figure 2.8. The topology distribution is presented in Figure 2.10. ResNet-18 is the most used topology followed by AlexNet. The rationale behind such distribution is that ResNet-18 offers a high accuracy in widely used applications such as image recognition and object detection [89]. As for AlexNet, it represents a small model (8 layers) that is capable of achieving high accuracy compared to larger CNNs [90]. Another fair share of network topology is the customized one. This is due to the fact that an existing CNN may not offer an acceptable performance for an application other the one trained for [86].

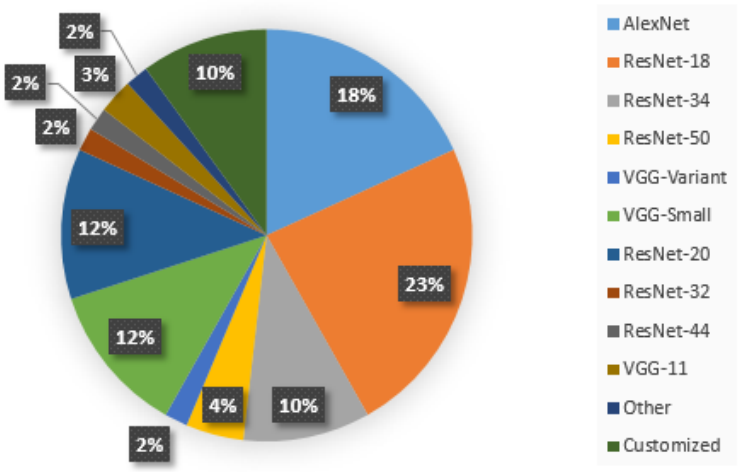


Fig. 2.10 Distribution of the most used Network Topology

2.3 Approximate Computing for Machine Learning Architectures

Approximate computing uses the statistical nature of data and algorithms to trade quality for savings. The savings are achieved using techniques that exploit the intrinsic resilience in machine learning applications to realize improvements in efficiency at all layers of the computing stack (algorithm, architecture, and circuit). At algorithmic level, approximations are applied on the data (e.g. sampling) and the functions that constitute the application software. At architecture level, the aim of the approximate computing techniques is mainly to replace a complex block with a relaxed version whose output is mathematically verified with an acceptable degree of inexactness. At circuit level, hardware designers focus on designing approximate versions of the most used arithmetic operations (e.g. adders and multipliers) which has the biggest influence in reducing the overall complexity.

Several research works have been performed to identify, define, and categorize the approximate computing techniques for better adoption and usage. Table 2.6 presents the reported approximate computing techniques, their definition, and where they have been mainly applied along the different parts of the computing stack. Next, we discuss the techniques that have been applied or investigated to be used for machine learning applications.

Quantization: also referred to as precision scaling, is a technique that changes the bit-width of input data or intermediate operands to reduce the computational and memory requirements. A 2-bit and 4-bit quantization has been adopted for the weights and activations of a CNN respectively in [91]. The quantized CNN achieved a higher performance of about 15× compared to full precision CNN. Within the context of low-precision fixed-point

computations, the research performed in [92] showed that deep neural networks can be trained using only 16-bit fixed-point number representation with stochastic rounding, and incur little to no degradation in the classification accuracy.

Loop Perforation: transforms loops to execute a subset of their iterations. The process identifies critical and tunable loops. The former can't be perforated due to unacceptable performance drop. The latter are loops whose perforation produces more efficient and generally acceptable accurate computations [93]. Loop perforation has been applied on several essential operations that are widely used in machine learning applications such as matrix multiplication, back propagation, kernel computation, etc [94].

The obtained results showed that performance can be enhanced up to 93%, while maintaining the quality loss at a rate below 10%.

Load Value Approximation: leverages the intrinsic nature of machine learning applications to estimate a value after its loading from a cache has been missed [95]. Such technique is adequate for machine learning implementations on both CPU and GPU. Yazdanbakhsh *et. al* proposed an approach to organize memory accesses and control flow to identify the loads that cause the largest fraction of misses, then approximating the ones leading to smaller degradation than a set threshold [96]. The proposed technique showed a performance and energy efficiency with bounded Quality of Result (QoR) loss in both the GPU and CPU.

Memoization: is a technique that stores the output of a function for later reuse under identical input trigger. Thus, some values can be approximated by reusing the results for similar functions. In machine learning applications, such technique can be applied by using "Approximate" values to increase the amount of successful value reuses [97].

Task/Memory Skipping and Pruning: similar to loop perforation, this technique skips a complete task within an algorithm or omit a memory access. For instance, an activation of a neuron is not performed if the weight value is under a certain threshold thus reducing both the memory storage requirements and the arithmetic computations of a CNN with a negligible effect on the classification accuracy [98]. The same methodology can be applied on the support vectors of an SVM kernel.

Data Sampling: this is one of the most used approximate computing technique in the machine learning domain, as it is even applied in the pre-processing phase. The idea is to reduce the the input size through Downsampling, Downscaling, dimensionality reduction, etc. Hence, affecting both the memory requirements and the computational complexity. Authors in [99] investigated the ability of data sampling to address the issue of class imbalance on two common neural network learning algorithms to improve their performance. In [56], the input tensor size is reduced from $4 \times 4 \times 30,000$ to $4 \times 4 \times 20$ without an accuracy loss leading to the ability to deploy the tensorial SVM algorithm on resource-limited platform.

Table 2.6 Approximate Computing Techniques applied at different stages of the computing stack

Technique	Definition	Computing Stack Level	Examples
Quantization	Changing the precision (bit-width) of input or intermediate operands	Algorithm/Architecture/Circuit	[91], [100], [92], [101]
Loop Perforation	Skipping some iterations of a loop	Algorithm	[93], [94], [102]
Load Value Approximation	Estimating load values to allow a processor to progress without stalling for a response after a load miss in a cache	Algorithm/Circuit	[103], [104]
Memoization	Storing the results of functions for later use with identical function call with the same input	Architecture	[105], [106], [107]
Task/Memory Skipping	Skipping memory accesses, tasks, or input portions	Algorithm/Architecture	[108], [109]
Data Sampling	Processing only a subset of input data through Downsampling or Downscaling	Algorithm	[110], [108], [111]
Using program versions of different accuracy	Utilizing multiple versions of an application code with multiple trade-offs between accuracy and overhead	Algorithm	[109], [112]
Inexact/faulty Hardware	Designing approximate adder, multiplier, and memory architectures	Algorithm/Architecture/Circuit	[113], [114], [115], [116]
Voltage Scaling	Reducing the supply voltage of certain parts (e.g SRAM)	Circuit	[117], [118], [119]
Use of Neural Networks	Replacing a complex block with a neural network producing the same mathematical output	Architecture	[120], [121], [111]
Synaptic Pruning	Reducing the number of weights synapses in a neural network	Algorithm	[122], [123]
Approximating Networks	Introducing approximate neurons or reducing the number of hidden layers	Algorithm	[124], [111]
Scalable Effort Design	Scaling the number of bits in data path between MAC and FIFOs	Architecture	[125], [126]

Use of Neural Networks: this technique aims at replacing a complex portion of an algorithm with a neural network. For example, authors in [127] have proposed a Multi-Layer Perceptron (MLP) NN-based accelerator for the approximation of widely used mathematical functions in a wide variety of machine learning algorithms such as exp, cos, etc. Compared to the conventional glibc (GNU C library) implementation, the MLP implementation achieved an energy-delay-product improvement of two orders of magnitude with negligible accuracy loss.

Approximate Networks: this is an extended version of the task/memory skipping technique where a complete artificial neural network (ANN) is approximated. In [111], authors proposed a novel approximate computing framework for ANNs. A 34.11% to 51.72% energy benefits with less than 5% quality loss have been recorded for various neural network applications using the proposed framework.

2.4 Conclusion

This chapter provides a compact survey on the learning algorithms used for tactile processing, the computational complexity and the hardware platforms that could be used for accelerating such algorithms. Then, the key techniques used for the design of kNN, SVM, and BCNN hardware accelerators are discussed. The techniques are presented in terms of their applicability, acceleration gain, and an example showing their deployment on different hardware devices targeting different applications. Then, a set of approximate computing methods is investigated. A definition and how-to apply such methods on machine learning algorithms are presented. Some existing solutions are highlighted to show the reductions offered by approximate computing techniques with respect to the introduced accuracy loss.

Chapter 3

Algorithmic Level Approximate Computing Techniques for Machine Learning

3.1 Introduction

Applications in domains like computer vision, media processing, machine learning, etc. have intrinsic tolerance to inaccuracy. Studies repeatedly show that such applications consist of both critical and non-critical components [128], [129], [130]. Thus, it is not necessary for every arithmetic operation to be precisely correct and every bit of memory to be preserved at the same level of reliability. From the perspective of approximate computing, not every operation in a program needs the same level of accuracy. Auto-tuning approaches can help empirically identify error-resilient components [131]. However, since machine learning algorithms don't present the same level of accuracy for all applications, the identification process varies from one application to another.

Approximate computing techniques (ACTs) on the algorithmic level have been firstly investigated in [132]. Authors introduced the concept of incremental refinement. Such concept aims at reducing the number of iterations of iterative processing. A factor of ten reduction in power consumption has been recorded for a speech finite impulse response (FIR) filter implementation. In [125], authors explored the different parameters of the Support Vector Machines (SVM) algorithm using algorithm level scaling to reduce the complexity of hardware implementations. A $1.2\times - 2.2\times$ energy savings have been achieved without any significant accuracy loss. Nogues *et. al* have presented an approach on how to apply algorithmic level approximate computing techniques on HEVC decoding [133]. Authors

offered a strategy on how to locate the error-resilient components of the decoder and which approximate technique to be applied. Energy reductions of up to 40% are demonstrated for a limited degradation of the application Quality of Service (QoS).

In this chapter, we propose an approach for applying algorithmic level approximate computing techniques on machine learning algorithms. The approach is based on the work presented in [133]. k-Nearest Neighbor (kNN) and Tensorial SVM (TSVM) algorithms are adopted for the evaluation of the proposed approach. The evaluation is performed on both software and hardware levels. For the software evaluation, we monitored the loss in classification accuracy of a touch modality problem presented in [5] with respect to the gain in execution time and memory requirements. The software evaluation has been accomplished on an Intel Central Processing Unit (CPU). For the hardware evaluation, a Field Programmable Gate Array (FPGA) implementation of both algorithms is presented. Then, the gain in hardware area, time latency, and power consumption is recorded when each technique is embedded in the hardware implementation. Results have shown that the kNN execution time and memory usage can be reduced up to 38% and 55% respectively. Similarly, a 29.6% power reduction and speedup up to 3.7× can be achieved with an approximate kNN FPGA implementation. As for approximate TSVM, the implementation achieves a reduction in power consumption by up to 49% with a speedup of 3.2×. All these reductions are accompanied with a classification accuracy loss than 10%.

3.2 Algorithmic Level Approximate Computing Techniques

Algorithmic level ACTs are divided into two categories: data-oriented and process-oriented. Figure 3.1 presents an approach on how to apply these techniques on machine learning algorithms. The data-oriented category involves modifying the data properties (size and bit-width) to minimize the work-load on the circuit level. This category includes:

- Dataset Reduction (DsR) decreases the amount of the processed data by eliminating samples randomly or using a sub-sampling method as the one proposed in [5]. Furthermore, DsR can be applied through Downsampling (DS) and Downscaling (DSc). The former adjusts the sampling frequency of the electronic interface used to collect raw data samples from sensors in the time domain, while the latter reduces the dimension of the collected data themselves (e.g., reducing the tensor size from $4 \times 4 \times 3$ to $3 \times 3 \times 3$), as shown in Figure 3.1.

- Data Format Modification (DFM) reduces the bit-width of the data and its corresponding arithmetic operations. This can be done by replacing floating-point representation

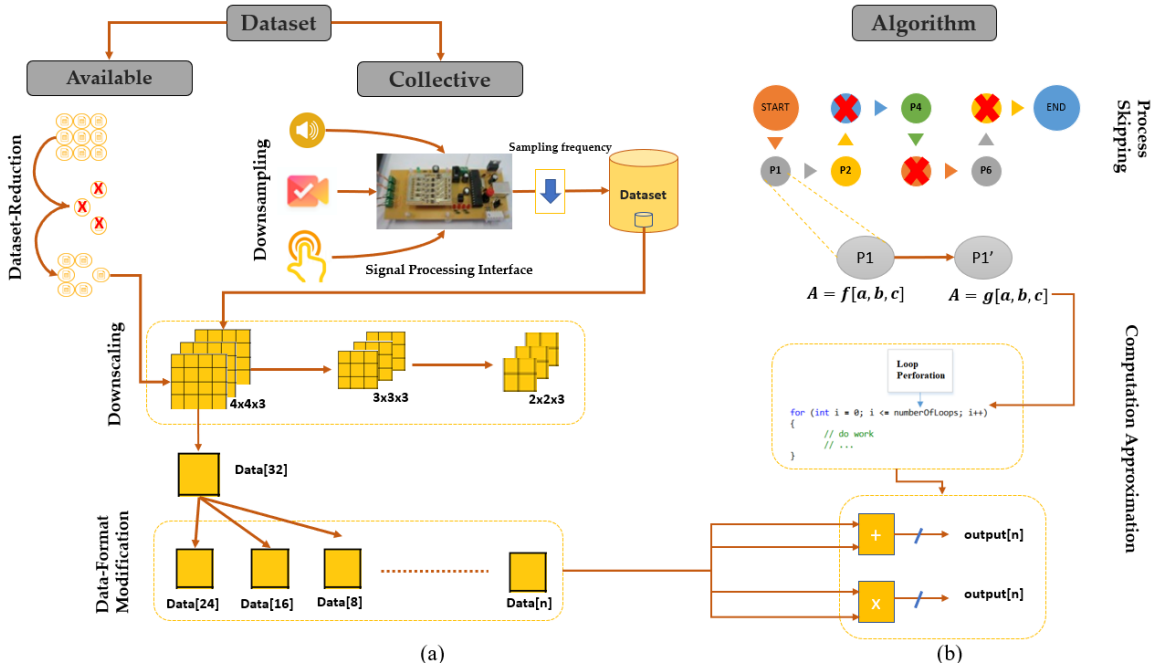


Fig. 3.1 Algorithmic level Approximate Computing Techniques: (**a**) data-oriented; and (**b**) processing-oriented.

with a fixed-point one. For instance, a 8-bit fixed-point representation data is adopted in [134] instead of floating-point to represent tactile data with a negligible precision loss.

The process oriented category targets the algorithm itself by reducing the number of operations or replacing some of them with a less-complex counterpart. This category includes [133]:

- Computation Skipping (CS) skips a certain number of operations in an algorithm. If these operations are loop iterations, then it is referred to as Loop Perforation (LP). For example, in some machine learning applications, a pre-processing operation such as data normalization may be skipped without affecting the quality of service of the target application.
- Computation Approximation (CA) proposes an equivalent version of a computationally complex function. The two versions should be mathematically equivalent with an acceptable output error margin. For example, a division function could be replaced by a reciprocal multiplication [133].

3.3 Experimental Setup

The ACTs presented in Figure 3.1 are evaluated on two machine learning algorithms, mainly kNN and TSVM. This section provides the details about the used dataset (for classification assessment), the software environment, and hardware environment used for the ACTs evaluation.

3.3.1 Dataset

A tactile dataset describing three touch modalities have been collected in [5]. The tactile data have been acquired by an electronic skin based on a piezoelectric sensor array. The dataset has been collected with the help of 70 participants. Each participant was required to preform one of the touch modalities on a 4×4 piezoelectric sensor: sliding the finger, brushing a paintbrush and rolling a washer as shown in Figure 4.5.

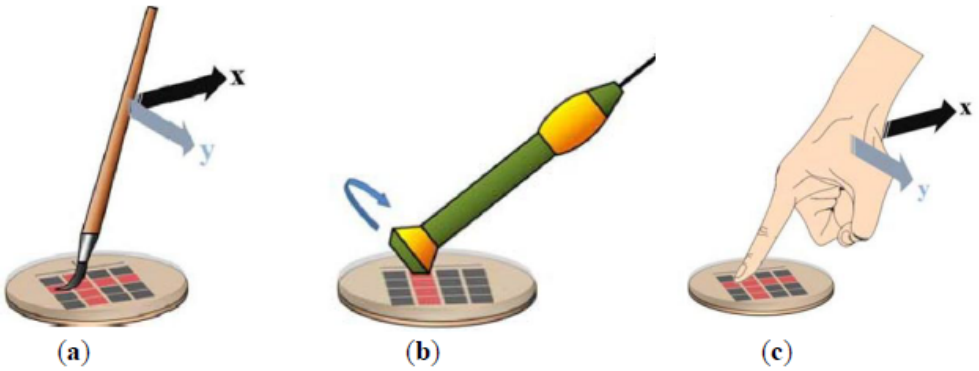


Fig. 3.2 Touch modalities. (a) Paintbrush brushing; (b) washer rolling (c) finger sliding;

Each participant performed a touch modality on both the horizontal and vertical directions for a duration of 10 seconds. Thus, with a sample rate of 3000 samples per second, the collected data can be expressed as a tensor $\phi(4 \times 4 \times 30,000)$. For complexity concerns, a sub-samnpling algorithm has been proposed to reduce the tensor size to $\phi(4 \times 4 \times D)$, where D is the sub-sampling ratio determined based on the required accuracy of the target application. This dataset can be used to formulate three binary classification problems:

- A: brushing a paintbrush versus rolling a washer
- B: brushing a paintbrush versus sliding the finger
- C: rolling a washer versus sliding the finger

These problems will be used to study the effect of applying the algorithmic level ACTs on kNN and TSVM algorithms.

3.3.2 Software Environment

Two classifiers for kNN and TSVM algorithms have been coded in C++ based on the architectures presented in [135] and [73], respectively. A performance profile is generated using Linux and Windows 10 running on a PC with Intel ® Core ™ i7-4510U x64 CPU clocked @ 2.6 GHz with a Random-Access Memory (RAM) of 12 GB. Each profile offers the execution time and memory usage of kNN classifier under various algorithmic level approximate computing techniques.

For our simulations, the execution time refers to the "user CPU time" which is the time spent on the processor running the classifier' code and libraries. Authors in [136] conducted a study about the challenges in time execution analysis. execution time depends on the cache, processor and the operating system management utilities. To reduce the effect of these components and obtain a credible execution time, we considered the following:

- The code of the classifier is the only application running while generating a profile.
- Each classifier is profiled alone to avoid overlap and resource consumption.

The execution time is obtained by averaging the time during five different runs with the use of two tools: *gprof* on Linux and Microsoft Visual Studio Debugger on Windows 10.

For a kNN classifier under different ACT, a memory usage profile has been generated by calculating the total memory allocations. These allocations belong to the training set, testing set, variables and functions' parameters. To obtain an accurate memory usage, the Linux tool *Valgrind* is used to ensure no memory leaks occurred and no unnecessary allocations were made.

3.3.3 Hardware Environment

The FPGA implementation of the kNN and TSVM architectures presented in [135] and [73] respectively is performed using High Level Synthesis (HLS). The architectures are modeled in C++ using Xilinx Vivado suite. Then, each architecture has been optimized using HLS directives and synthesized to ensure that it fits in the target FPGA device. Then, a C/RTL simulation was performed to ensure a coherent output from the architecture coded in C++ and the RTL design provided by Vivado HLS. Afterward, each architecture has been exported as an RTL IP block targeting a Zynq-7010 and Virtex-7 XC7VX980T FPGA for

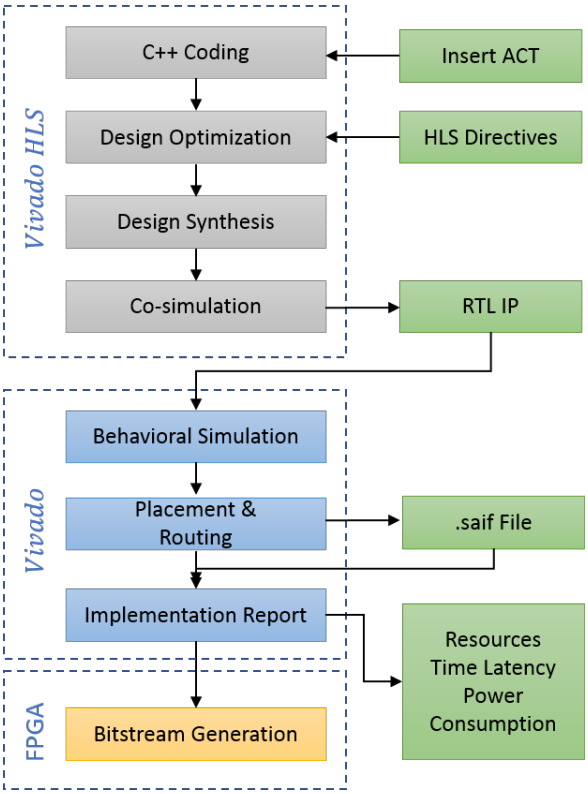


Fig. 3.3 FPGA Implementation Process

kNN and TSVM implementations respectively operating at a clock frequency of 120 MHz. The IP block has been imported into Vivado; then, a behavioral/combinational simulation is performed to verify the integrity of the exported IP. Then, place and route was performed to implement the architecture on the FPGA device. Finally, a detailed report about the utilized hardware resources, the number of clock cycles and the power consumption are obtained once the implementation is completed. The implementation details can be summarized as shown in Figure 3.3.

3.4 Approximate k-Nearest Neighbor

3.4.1 kNN Overview

kNN is a classification algorithm that assigns a class to a query object by a majority of vote k. The most complex operations involved in kNN classification are: Distance calculation and Sorting Process. Since, kNN doesn't create a model for training, every classification task involves calculating the distance between the query object and all training objects. Hence, our main focus is to reduce the data size, consequently reducing the number of distance

calculations to be performed. For this purpose, the data-oriented approximate computing techniques are adopted.

3.4.2 Software Simulation

For our knowledge, a kNN classifier hasn't been used for the touch modality classification problem presented in section 3.3.1. Thus, this section details the evaluation process of both Exact and Approximate kNN.

Exact kNN: The main parameter to be tuned in kNN is the number of nearest neighbors k. To find the value of k, the tactile dataset has been used in four scenarios:

- 80% training, 20% testing.
- 85% training, 15% testing.
- 90% training, 10% testing.
- 10-fold Cross Validation (CV): This process divides the dataset into ten equal subsets. Each subset is considered as a testing test for a single run i. The classification accuracy is calculated as:

$$Accuracy = \frac{1}{10}\sum_{i=1}^{10} Accuracy_i \tag{3.1}$$

The value of k has been varied between 1 and 10. Tables 3.1 and 3.2 presents the highest classification accuracy that kNN can achieve under different scenarios. The obtained results show that the touch modality classification is a challenging task especially for Problems A and B. This is evident in the variation of the classification accuracy percentage which differs from one scenario to another. According to the authors in [5], the data collection protocol used might be a reason along with the presence of a level of overlap between stimuli that in principle belonged to different touch modalities.

Table 3.1 Classification Accuracy under scenarios A, B, and C

Problem	**Classification Accuracy (%)**								
	Scenario A			**Scenario B**			**Scenario C**		
	k=3	**k=5**	**k=7**	**k=3**	**k=5**	**k=7**	**k=3**	**k=5**	**k=7**
A	51.78571	51.78571	48.21429	54.76191	66.66666	71.42857	64.28571	60.71429	64.28571
B	67.85714	76.78571	75	69.04762	76.19045	73.80952	67.85714	67.85714	64.28571
C	83.9285	80.35714	78.57143	85.71429	83.33334	80.95238	82.14286	82.14286	82.14286

The obtained results could be interpreted as follows:

Table 3.2 Classification Accuracy under scenario D

Run	Classification Accuracy (%)								
	Problem A			Problem B			Problem C		
	k=3	k=5	k=7	k=3	k=5	k=7	k=3	k=5	k=7
1	64.28571	60.71429	64.28571	67.85714	67.85714	64.28571	82.14286	82.14286	82.14286
2	75	78.57143	78.57143	71.42857	96.42857	85.71429	89.28571	85.71429	82.14286
3	57.14286	50	57.14286	78.57143	78.57143	82.14286	75	75	75
4	78.57143	71.42857	71.42857	82.14286	78.57143	85.71429	92.85714	92.85714	92.85714
5	39.28571	53.57143	42.85714	89.28571	96.42857	96.42857	78.57143	78.57143	75
6	82.14286	82.14286	82.14286	50	50	50	92.85714	92.85714	92.85714
7	50	53.57143	60.71429	32.14286	25	32.14286	100	100	100
8	89.28571	82.14286	82.14286	60.71428	50	57.14286	96.42857	96.42857	100
9	71.42857	75	82.14286	50	50	46.42857	89.28574	89.28571	89.28571
10	75	71.42857	67.85714	57.14286	57.14286	53.57143	100	100	100
CV	68.21429	67.85714	68.92857	63.92857	65	65.35714	89.64286	89.28571	88.92857

- For Problem A: The classifier suffered from a relatively high classification error percentage, which at its best reached 28% for 85% splitting ratio with k=7. The general effect of changing the number of neighbors k can't be generalized effectively. For example, in Scenario B: as k increases the classification error decreases which is not the case for Scenario A or C. The highest classification accuracy obtained is approximately 69% while using 10-fold cross validation and k=7.

- For Problem B: The classifier showed a better result than those of Problem A, which at its best reached a classification error percentage of 23% for 80% splitting ratio and k=5. The general effect of changing the number of neighbors k can't be generalized effectively. For example, in Scenario A: as k increases the classification error increases which is not the case for Scenario B or C. The highest classification accuracy obtained is slightly greater than 66% while using 10-fold cross validation with k=7.

- For Problem C: The classifier showed a better result than both of Problems A and B, which at its best reached a classification error percentage of 14% for 85% splitting ratio and k=3. As the number of nearest neighbors k increases the classification error percentage increases for Scenarios A and B, while it didn't affect the classifier performance for Scenario C. The highest classification accuracy obtained is around 90% while using 10-fold cross validation with k=3.

Comparing the obtained results with those obtained in [5]: for Problems A and B, the tensorial SVM classifier scored a lower classification error percentage than the KNN classifier with a relative difference of 5%-10%. While for Problem C, the KNN classifier achieved a classification accuracy increase of 15% with $k = 3$. Hence, the data oriented approximate

computing techniques will be investigated on Exact kNN with $k = 3$ targeting the touch modality classification problem C.

– *Approximate kNN:* The Exact kNN has been modified to include DsR, DFM, and *"cross-layer"* (CL) approximate computing techniques. Cross-layer approximation involves the combination of two or more techniques. Dataset reduction is applied through Downscaling (10%, 20%, and 30%). Downscaling is applied by reducing each touch modality sample from $\phi(4 \times 4 \times 30,000)$ to $\phi(4 \times 4 \times D')$, where D' is the number of readings remained after 10% (or 20%, 30%) reduction using the sampling algorithm proposed in [5]. As for data format modification, 24-bit and 16-bit fixed-point representations were applied for all the kNN operations with <8,16> and <6,10> precision, respectively, using the C libraries used in [137]. Downscaling and DFM are combined and denoted by *CL1* and *CL2* i.e the combination of 24-bit fixed-point with Downscaling and 16-bit fixed-point with Downscaling respectively.

Table 3.3 shows the effect of Downscaling on kNN classification accuracy for Problem C. For Scenario A, DSc up to 20% shows no accuracy loss while a DSc of 30% only shows a 1.7% loss. The same behaviour is noticed for Scenario C, but with a loss of 7.14% at 30% DSc. For Scenario B, Downscaling has no effect on the classification accuracy. For Scenario D, as the the Downscaling percentage increases, the loss increases with a threshold less than 6%. The obtained results show that kNN achieves a high classification of touch modalities even if the dataset is downscaled up to 20%.

Table 3.3 Effect of Downscaling on kNN classification accuracy

Approximate Computing Technique	**Classification Accuracy (%) for Problem C, k=3**			
	Scenario A	**Scenario B**	**Scenario C**	**Scenario D**
None (Exact kNN)	83.92	85.71	82.14	89.64
10% Downscaling	83.92	85.71	82.14	88.49
20% Downscaling	83.92	85.71	82.14	87.05
30% Downscaling	82.14	85.71	75	83.67
Loss (%) after 10% DSc	0	0	0	1.15
Loss (%) after 20% DSc	0	0	0	2.58
Loss (%) after 30% DSc	1.78	0	7.14	5.96

Table 3.4 shows the effect of the transition from floating-point to fixed-point representation on kNN classification accuracy. For all scenarios, 24-bit representation lead to a higher classification accuracy than 16-bit representation. As the testing set size decreases (from Scenario A to B to C), the accuracy loss decreases for both fixed-point representations with a best case of less than 4% loss in for a 90% dataset split and 24-bit representation.

Table 3.4 Effect of Data Format Modification on kNN classification accuracy

Approximate Computing Technique	**Classification Accuracy (%) for Problem C, k=3**			
	Scenario A	**Scenario B**	**Scenario C**	**Scenario D**
None (Exact kNN)	83.92	85.71	82.14	89.64
24-bit Data Format Modification	73.21	78.57	78.57	86.07
16-bit Data Format Modification	62.5	64.28	71.42	79.28
Loss (%) after 24-bit DFM	10.7	7.14	3.57	3.57
Loss (%) after 16-bit DFM	21.42	21.42	10.71	10.35

Table 3.5 shows the kNN classification accuracy when Downscaling and DFM are used at the same time. At first glance, the effect of cross-layer approximation is not uniform. This is expected since a query that is incorrectly classified due to Downscaling may or may not have been incorrectly incorrectly classified due to DFM. As the testing set size decreases (from Scenario A to B to C), the accuracy loss decreases for both *CL1* and *CL2*. This can be justified due to the fact that Scenario C lead to the lowest accuracy loss when DSc and DFM has been adopted individually. Such loss is consistent regardless of the Downscaling percentage. A kNN classification with 30% *CL1* achieves the lowest accuracy loss of 3.57%. Such loss increases dramatically to more than 10% for *CL2* due to the effect of 16-bit fixed-point representation as reported in Table 3.4.

Table 3.5 Effect of Cross Layer Approximate Computing on kNN classification accuracy

Approximate Computing Technique	**Classification Accuracy (%) for Problem C, k=3**			
	Scenario A	**Scenario B**	**Scenario C**	**Scenario D**
None (Exact kNN)	83.92	85.71	82.14	89.64
10% Cross Layer 1	75	78.57	78.57	84.52
20% Cross Layer 1	66.07	78.57	78.57	83.03
30% Cross Layer 1	66.07	73.8	78.57	79.59
Loss (%) after 10% CL1	**8.92**	**7.14**	**3.57**	**5.12**
Loss (%) after 20% CL1	**17.85**	**7.14**	**3.57**	**6.61**
Loss (%) after 30% CL1	**17.85**	**11.91**	**3.57**	**10.05**
10% Cross Layer 2	62.5	69.04	71.42	76.98
20% Cross Layer 2	62.5	64.2	71.42	76.78
30% Cross Layer 2	60.71	64.2	71.42	74.4
Loss (%) after 10% CL2	**21.42**	**16.67**	**10.72**	**12.66**
Loss (%) after 20% CL2	**21.42**	**21.51**	**10.72**	**12.86**
Loss (%) after 30% CL2	**23.21**	**21.51**	**10.72**	**15.24**

–Profiling of Approximate kNN: To assess the advantages of algorithmic level approximate computing techniques, their use is profiled in terms of execution time and memory usage

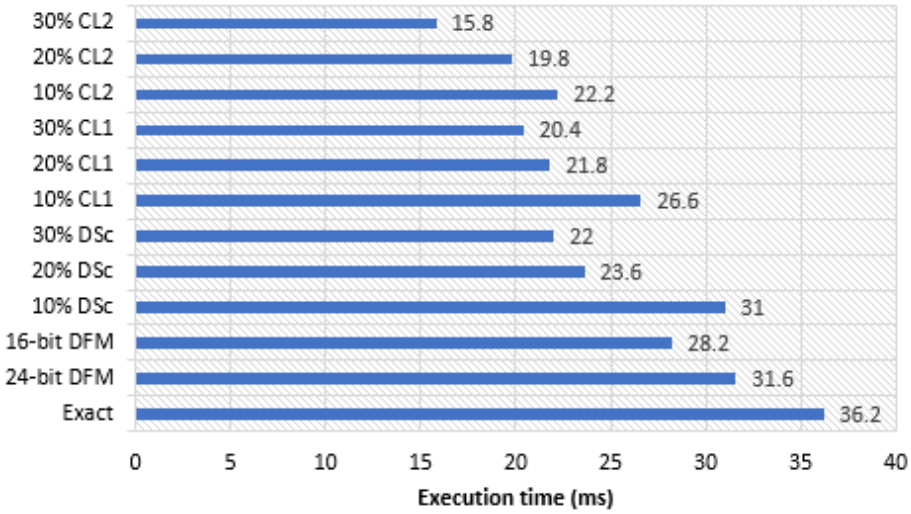


Fig. 3.4 Execution Time of a kNN Classifier under various ACTs

while running on an Intel CPU. Figure 3.4 presents the execution time recorded as an average over 5 runs of Exact and approximate kNN under the discussed techniques. The lowest execution time is achieved using 30% *CL2*, as it involves the highest Downscaling percentage accompanied with the lowest fixed-point representation used i.e. 16-bit. As for Downscaling versus DFM, the effect varies on their adoption. For example, a kNN classifier with 16-bit DFM achieves a touch classification in 28.2 ms compared to 31 ms using 10% DSc. However, for higher Downscaling percentages (e.g. 20%), a lower execution time is recorded. For 24-bit DFM, a kNN classifier with Downscaling always achieves a lower execution time.

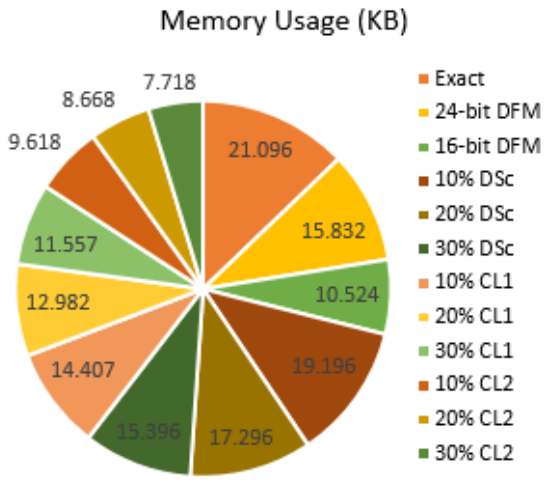


Fig. 3.5 Memory Usage for a kNN Classifier under various ACTs

Figure 3.5 reports the required memory usage in Kilo Bytes (KB) for the adoption of each ACT. The results show that DFM requires less memory usage compared to Downscaling. Thus, for the touch modality problem, reducing the data bit-width has a more evident effect on reducing the memory usage than Downscaling the data size. As expected, the lowest memory usage is recorded for a KNN classifier with 30% *CL2*, due to the lowest fixed-point representation and highest Downscaling percentage.

The timing and memory requirements differ from one application to another based on the available budget. Figure 3.6 plots the classification accuracy accompanied with the reduction in execution time and memory usage for each ACT. This plot can be used as a reference to determine the best adequate techniques for a certain set of requirements. A few observations can be noted:

- For a classification accuracy of 71%, the 30% *CL2* should be adopted instead of less efficient techniques such as 16-bit DFM or 10% *CL2*.
- To obtain the highest classification possible, it is advised to use a Downscaling percentage less then 30%.
- For a moderate accuracy loss, the *CL1* techniques is the best fit.

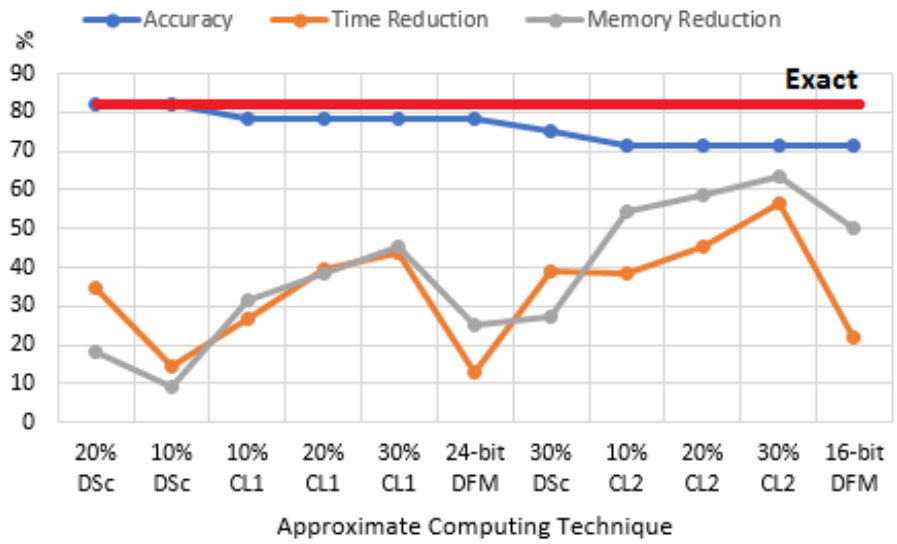


Fig. 3.6 kNN Performance Profile under various ACTs

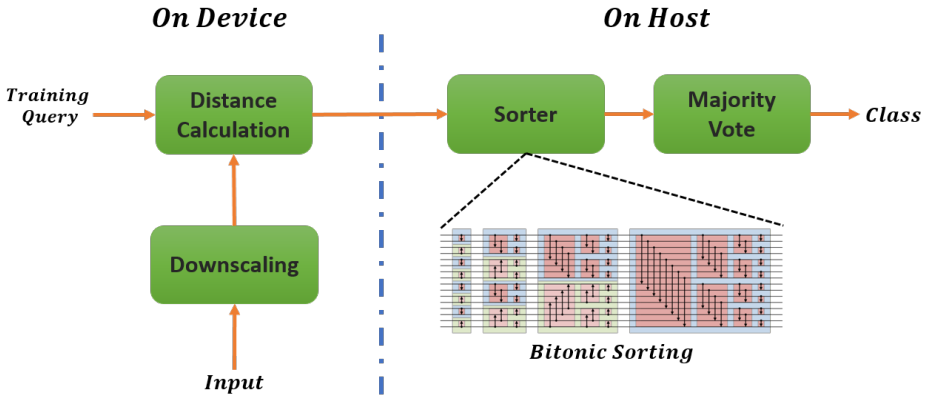


Fig. 3.7 Approximate kNN Architecture

3.4.3 Hardware Implementation

Figure 3.7 presents the kNN architecture adopted from [135] for the hardware implementation targeting the Zynqberry platform. The architecture adopts a hybrid approach to accelerate the kNN implementation. The distance calculation is performed on FPGA referred to as *on Device*, while the sorting process and majority voting blocks are executed on CPU referred to as *on Host*. The squared Euclidean distance and Bitonic have been chosen for distance calculation and sorting blocks. The Majority vote block assigns a class for the input based on the class with the highest number of occurrence in the sorted vector.

3.4.4 Implementation Results and Assessment

Table 3.6 details the implementation resources of the Exact and approximate kNN under different ACTS. The recorded resources include the hardware area (BRAM, DSP, etc.), time latency, and power consumption. The time latency (L) is recorded as:

$$L = N \times \frac{1}{f_{max}} \tag{3.2}$$

where N is the number of clock cycles and f_{max} is the maximum operating frequency. As for the power consumption, a vector-based method was adopted as it provides the power consumption related to the processing under a defined testbench. The method involves generating a "saif" file via post-implementation functional and timing simulations.

The obtained results show that kNN is capable of real-time classification of touch [4] modalities in 0.56 ms with a power consumption of 27 mW. The real-time processing is an essential requirement for applications such as prosthesis, robotics, and industrial packaging, etc. Such results are further enhanced with the use of data-oriented approx mate computing

Table 3.6 kNN Implementation Report targeting Zynqberry operating at 120 MHz

Approximate Computing Technique	Implementation Resources						
	BRAM	DSP48E	FF	LUT	Slice	Time Latency (ms)	Power Consumption (mW)
None (Exact kNN)	4	5	709	841	293	0.56	27
24-bit DFM	4	4	272	425	150	0.15	19
16-bit DFM	2	1	205	244	93	0.12	11
10% DSc	4	5	709	841	293	0.51	25
20% DSc						0.45	24
30% DSc						0.39	23
10% CL1	4	4	272	425	150	0.14	17
20% CL1						0.12	16
30% CL1						0.11	15
10% CL2	2	1	205	244	93	0.11	9
20% CL2						0.09	8
30% CL2						0.08	7

techniques which is evident in the reduced time latency and power consumption. The reported hardware resources are decreased with the use of DFM but not with Downscaling. The reason is that DSc is applied offline on the input only. While DFM is applied on all the kNN blocks. Hence, the hardware resources are not reduced with the use of cross-layer approximate techniques. As expected, the fastest classification time and lowest power consumption are recorded with the use of *CL2*.

To highlight the advantages of using algorithmic level approximate computing techniques on the kNN implementation, a plot for the degradation of accuracy with respect to the reduction in power consumption and classification speedup is presented in Figure 3.8.

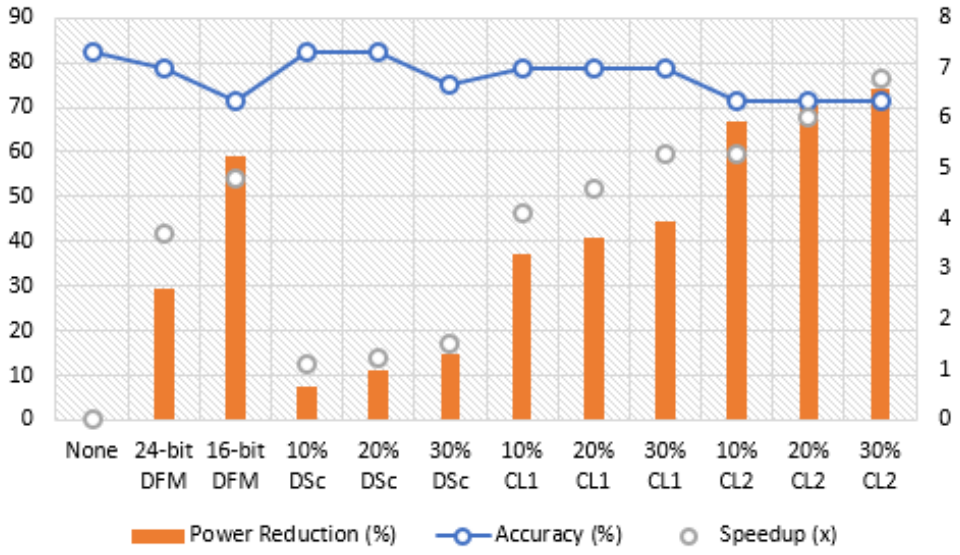


Fig. 3.8 kNN FPGA Implementation Performance under various ACTs

Using this plot, one could select the best ACTs for approximate kNN based on the applications' requirements. For example, an approximate kNN with *CL2* offers speedup up to 6.8× with 74% power reductions at the expense of 10% accuracy loss. For an acuracy loss of 6%, an approximate kNN with *CL1* offers speedup up to 5.5× and about 45% power reductions. Hence, based on the error tolerance of an application, a corresponding approximate computing techniques could be used to achieve huge gains i terms of time latency and power consumption.

3.5 Approximate Tensorial Support Vector Machine

3.5.1 Tensorial SVM Overview

SVM classification of an input tensor has been introduced in a framework that extends kernel methods to tensor data in 4 main steps [5]:

- Tensor Unfolding: A tensor $\phi(I_1 \times I_2 \times I_3)$ is transformed into three matrices $X_1(I_1 \times I_2 I_3)$, $X_2(I_2 \times I_1 I_3)$ and $X_3(I_3 \times I_1 I_2)$.
- Singular Value Decomposition (SVD) Computation: The unfolded matrices are symmetrized into square matrices that can be written in the form:

$$X = USV^T \tag{3.3}$$

  where U and V^T contain the left and right singular vectors respectively, and S is the diagonal matrix storing the singular values σ_i of X.

- Kernel Computation: The tensorial kernel extended from the Gaussian kernel is computed using the function:

$$K(x,y) = \prod_{1}^{z} k^z(x,y) \tag{3.4}$$

  where k^z is the kernel factor defined as:

$$k(x,y) = \exp\left(\frac{-1}{2\sigma^2}(I_n - trace(Z^T Z))\right) \tag{3.5}$$

  where $Z = V_x^T V_y$, V_x and V_y represent the singular vectors of the unfolded matrix obtained during the inference and training phase respectively, and trace represents the sum of diagonal elements.

- Classification: Applying the SVM classification function expressed as:

$$\hat{y} = f_{SVM}(x) = \sum_{i}^{n} \beta_i K(x_i, x) + b \tag{3.6}$$

where $\hat{y}$ is the predicted label of input tensor x, n is the number of training tensors, β_i are the coefficients obtained during training, and b is the bias.

3.5.2 Software Simulation

The touch modality classification problems presented in section 3.3.1 are used for accuracy evaluation. The tactile dataset has been split into 70% training and 30% testing sets. The dataset reduction is applied by randomly removing samples from the original dataset. To ensure credible assessment, if a sample is removed during the 10% reduction, it is automatically removed for the 20% and 30% reductions. Loop perforation is applied on the loops of SVD in (3.3) with a skipping factor sf (i.e., how many loops are perforated). As for data format modification, 24-bit and 16-bit fixed-point representations are applied for all the TSVM operations with $< 8, 16 >$ and $< 6, 10 >$ precision, respectively, using the C libraries used in [137].

Table 3.7 presents the effect of each ACT on the classification accuracy of the TSVM. Results show that 10% DsR doesn't affect the accuracy of TSVM for the three classification problems. Once the DsR percentage increases to 20%, the accuracy drops with a worst case scenario of 10% drop for Problem A. When applying loop perforation, the accuracy drops with the increase of the skipping factor for problems A and C. For problem B, skipping more than two loops showed no effect on the classification accuracy. As for DFM, the TSVM showed the highest accuracy loss compared to other methods for the three problems when 16-bit fixe-point representation is used. Such loss could be dropped to 5% by adopting the 24-bit representation.

3.5.3 Hardware Implementation

Figure 3.9 shows the architecture of the approximate TSVM that is adopted to evaluate the use of algorithmic level ACTs for hardware implementation. The architecture of the approximate TSVM can be described through: *Offline Training*, *Online Inference*, and *Performance Booster*.

–Offline Training: The training process starts by activating the AU1 (Approximate Unit 1) (see Figure 3.9). AU1 applies dataset-reduction through DSc technique on the dataset by performing the following steps:

Table 3.7 Effect of approximate computing techniques on TSVM classification accuracy

Approximate Computing Technique	**Classification Accuracy (%)**		
	Problem A	**Problem B**	**Problem C**
None (Exact TSVM)	90.47	80.95	78
10% Data Set Reduction	90.47	80.95	78
20% Data Set Reduction	80.95	75	70
30% Data Set Reduction	80.95	75	70
Loop perforation with sf = 2	90.47	80.95	78
Loop perforation with sf = 3	85.71	75	65
Loop perforation with sf = 4	80.95	75	62.5
DFM (24-bit)	85.71	75	73
DFM (16-bit)	75	65	62.5

- The dataset size is reduced by eliminating data that corresponds to five participants with noisy readings. Figure 3.10(a) shows an example of such reading where the voltage is almost constant along the measured time. Therefore, the machine learning model will not learn new information from such sample. Hence, it is removed.

- During data collection of the tactile dataset, no precise instructions were given to the participants regarding the amount of pressure to be applied on the sensor [5]. Thus, some touch samples with silent readings where observed such as the one presented in Figure 3.10(b). Such samples could be pre-processed to extract meaningful information in certain time frame. Each sample is truncated from 10 to 3.3 s by omitting readings outside the interval [3.7, 7] s. This results in a new tensor $\phi(4 \times 4 \times 10,500)$.

- To reduce the computational complexity of the tensor-based learning algorithms, the tensor size could be reduced without the loss of information originality using sub-sampling. The latter is applied by truncating each sample into a new tensor $\phi(4 \times 4 \times 40)$ with 40 random time readings.

Then, the resulting tensor is unfolded into three matrices $M(4 \times 160)$, $N(4 \times 160)$ and $P(40 \times 16)$ that have to be symmetrized before applying SVD. The resulting support vectors along with the Gaussian parameter $\sigma = 1$ are fed to the kernel computation block (see Figure 3.9). The block outputs the kernel matrices for $(+1$ vs. $-1)$, $(+1$ vs. $+1)$, $(-1$ vs. $+1)$ and $(-1$ vs. $-1)$ binary classification problems where each row being labeled with the corresponding class label. This step is essential since LIBSVM [138] does not support tensorial kernels by default but can receive precomputed kernels. The LIBSVM library is used to obtain a classification model based on the precomputed kernel. The model contains the coefficients β_i and the bias b.

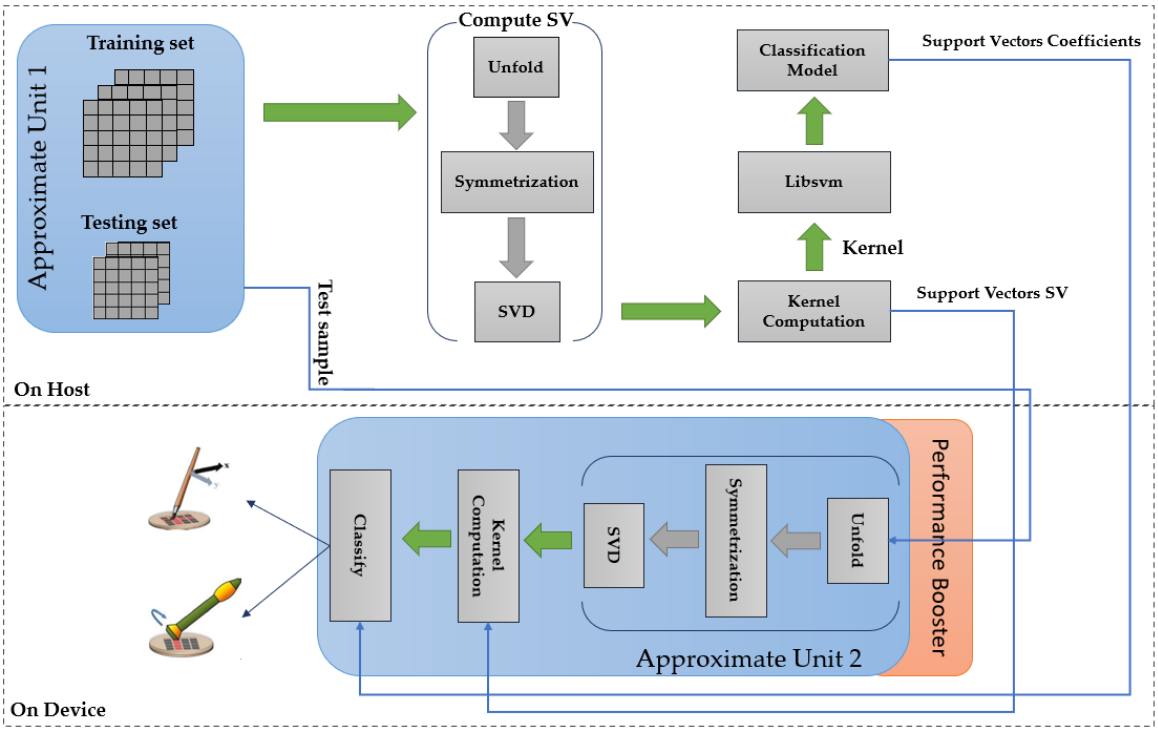


Fig. 3.9 Approximate TSVM Architecture

–*Online Inference:* The inference starts by fetching a sample tensor from the testing set (that was already approximated using AU1). The selected tensor undergoes the unfolding, symmetrization and SVD processes. The obtained support vectors along with the support vectors from the training phase are provided to the kernel computation block. During online inference, Approximate Unit 2 is active. It operates by applying:

- Loop perforation technique to the SVD block. The support vectors are obtained using the one side Jacobi Algorithm [139]. The latter is an iterative algorithm, thus it is perforated with $sf = 2$. This technique accelerates the SVD computations but a large sf could not be applied to ensure the algorithm's convergence.

- Computation Approximation to the computation of Z in (3.5). The obtained singular vector matrices from the SVD block are $V1(160 \times 160)$, $V2(160 \times 160)$ and $V3(16 \times 16)$. These matrices are truncated to $V1'(160 \times 4)$, $V2'(160 \times 4)$ and $V3'(16 \times 2)$. Such truncation reduces the complexity of the matrix multiplication in (3.5) with an acceptable error margin. This technique was also applied in the offline training phase so that the equation $Z = V_x^T V_y$ has correct dimensions.

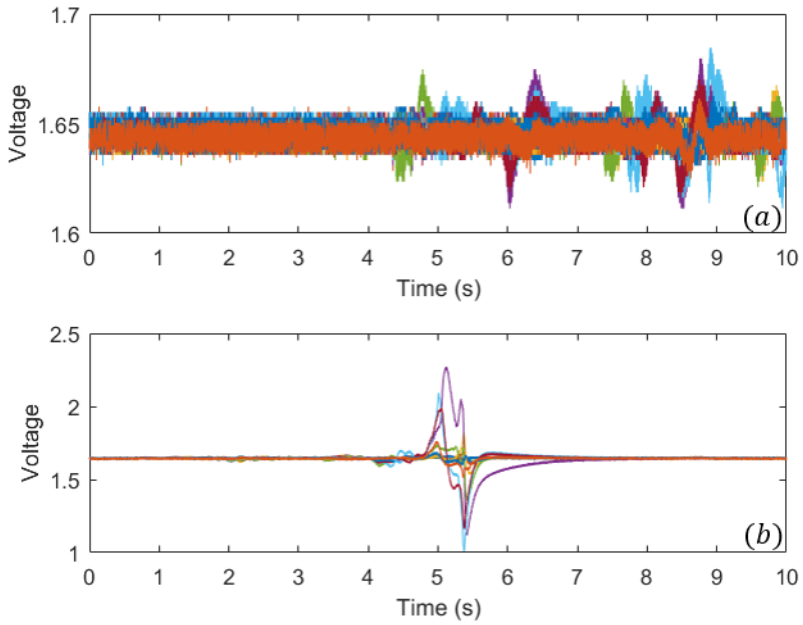


Fig. 3.10 Touch Modalities: (**a**) touch with noisy readings; and (**b**) touch with silent intervals.

- Data Format Modification to all the variables and arithmetic operations in different TSVM blocks. HLS offers a library called "apfixed", which allows the declaration of variables with fixed-point precision. This declaration is limited by an upper bound [140]. Specifically, the mathematical functions are Square Root (*sqrt*), which is used in SVD calculations, and Exponential (*exp*), which is used in kernel computations. These functions are supported only for bit-widths $w \leq 32$ and $w \leq 16$, respectively. This limitation was resolved by a variable precision architecture. Hence, all the inference blocks are implemented with 24- bit fixed-point representation with a $< 12, 12 >$ precision except the kernel computation block.

Finally, the output of the kernel computation (i.e., a kernel) is used by the classification block to predict a class for the tested tensor according to (3.6).

–Performance Booster: The performance of the proposed approximate TSVM architecture has been enhanced to achieve the lowest possible time latency for applications with timing constraints [141] while increasing the throughput. These requirements are usually accompanied by an increase in hardware resources, but the use of algorithmic level approximate computing techniques, specifically DsR and DFM, would compensate such increase.

The time latency and throughput requirements are facilitated by the use of Vivado HLS optimization directives [140]. The used directives are:

- Array Partition: This directive partitions a large Block RAM (BRAM) occupied by a multidimensional array into smaller separate memories. The array partitioning can be complete, cyclic or block. The latter was applied on the tensor $\phi(4 \times 4 \times 40)$ with block size = 16, as shown in Figure 3.11. This results in an RTL IP block with smaller memories while improving the throughput of the Unfolding process.

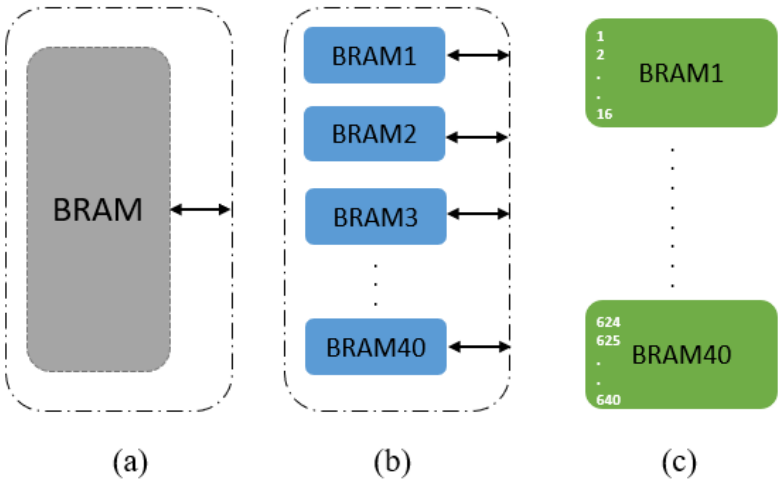


Fig. 3.11 Array partitioning: (**a**) without partitioning; (**b**) block partitioning; and (**c**) block with size=16.

- Dataflow: This directive allows functions to overlap in their operations, enhancing the overall throughput and latency of the design. The functions unfold and symmetrization are executed in a task-level pipelining using this directive, as shown in Figure 3.12.
- Pipelining: This directive allows the parallel execution of loop iterations, hence reducing the time latency. The computation of Z in (3.5) is executed in parallel, as shown in Figure 3.13.

3.5.4 Implementation Results and Assessment

Figure 3.14 shows the normalized speedup and reduction in power consumption while assessing different approximate computing techniques. The latter are applied one-by-one resulting in eight different FPGA implementations.

Using the obtained results in Figure 3.14, a cross-layer approximate TSVM implementation was performed where the adopted techniques are: 10% dataset reduction, loop perforation

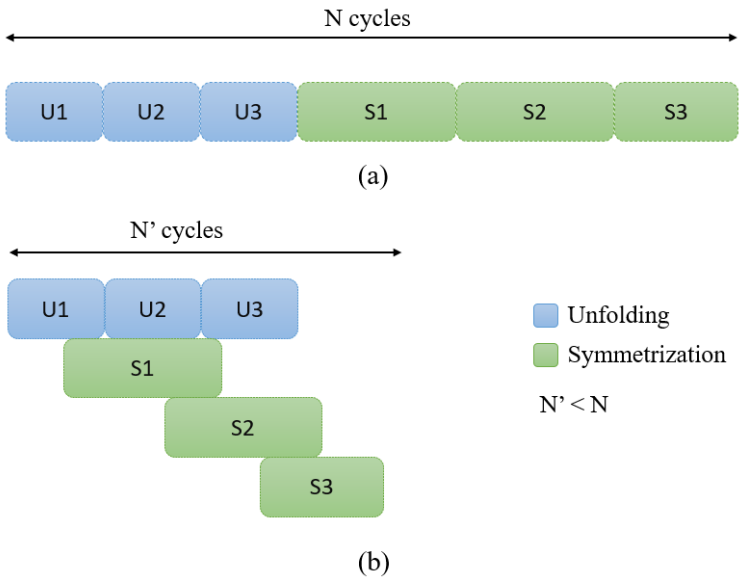


Fig. 3.12 Process: (**a**) without dataflow; and (**b**) with dataflow.

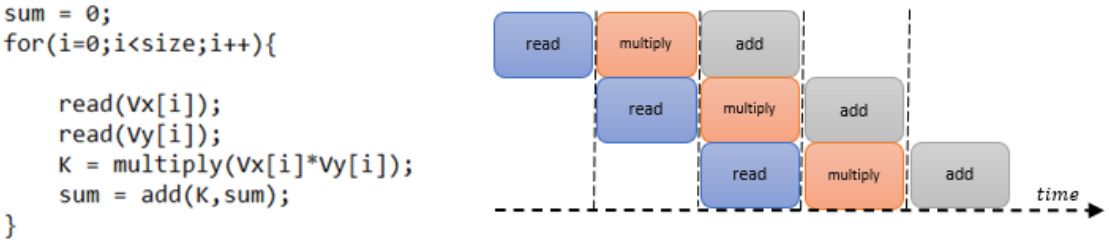


Fig. 3.13 Pipeline directive applied on vector multiplication.

with $sf = 2$ and 24-bit DFM. Moreover, the implementation details was recorded with the performance booster ON and OFF to differentiate between the gain due to approximate computing techniques with and without HLS optimization directives. Table 3.8 summarizes the performance profile for the FPGA implementations based on the architecture in Figure 3.9. The Exact SVM is based on the architecture presented in [56]. The boosted approximate TSVM corresponds to the approximate TSVM where the "Performance Booster" block is activated (See Figure 3.9), i.e., with HLS optimization directives. The reduction is calculated as:

$$Reduction(\%) = 100 - (\frac{I_{approx}}{I_{exact}} * 100) \tag{3.7}$$

where I_{approx} and I_{exact} are the implementation element (FF, DSP, LUT, etc.) of the Approximate (or boosted approximate) and Exact TSVM, respectively. As for the energy per

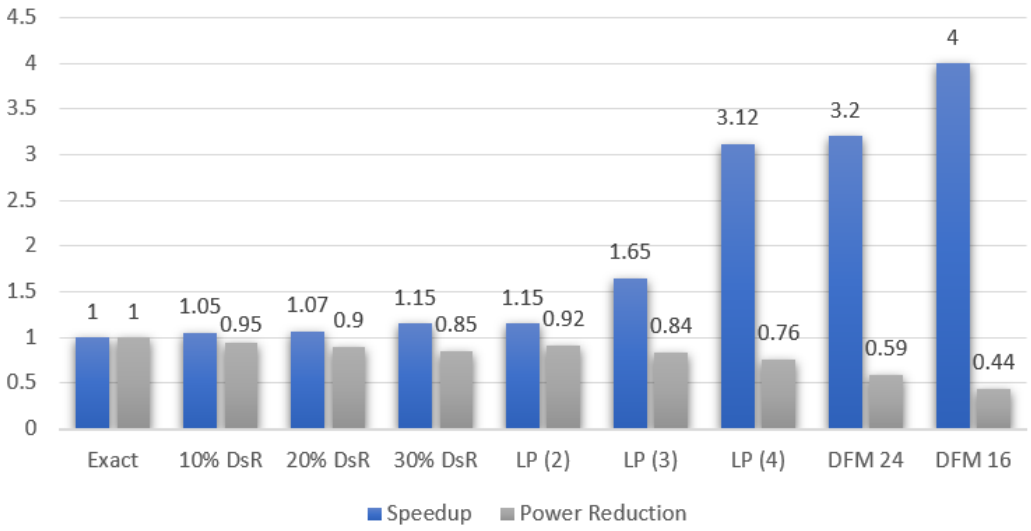


Fig. 3.14 Speedup and power consumption reduction under different ACTs

classification, it is calculated using the equation:

$$E = P \times T \tag{3.8}$$

where T is the time latency and P is the dynamic power consumption reported in Vivado.

Table 3.8 FPGA performance profile of Exact and Approximate TSVM

Architecture	FF	LUT	DSP	BRAM	SRL	Time Latency (s)	Power Consumption (W)	Energy per Classification (J)	Classification Accuracy (%)
Exact TSVM	37057	42261	475	297	1060	2.4	6.3	15.12	90
approximate TSVM	17,187	25,558	283	291	202	0.91	3.12	2.83	86
Boosted approximate TSVM	17,197	25,588	284	292	203	0.75	3.2	2.4	86
Approximate to Exact Reduction	53.62%	39.5%	40.4%	2.02%	80.94%	2.64×	50.4%	81.28%	−4%
Boosted Approximate to Exact Reduction	53.59%	39.45%	40.21%	1.68%	80.84%	3.2×	49.2%	84.12%	−4%

The obtained results presented in Tables 3.7 and 3.8 and Figure 3.14 demonstrate the effectiveness of using approximate computing techniques to reduce the hardware resources utilization, time latency and power consumption of the FPGA implementation of the tensorial SVM. Such reductions are accompanied by an accuracy loss that varies between 0% and 10%. Another set of remarks can be noticed:

- In general, loop perforation achieves lower latency and power consumption compared to dataset reduction with a comparable accuracy loss. This can be justified since the SVD computation block is among the most complex blocks of the tensorial SVM, as reported in [73].

- The transition to fixed-point representation results in the lowest latency and power consumption compared to other methods. This is expected due to the reduced complexity of the arithmetic operations based on fixed-point representation. This can be seen in the reduced number of required DSPs between the exact and approximate implementations. However, this comes at the expense of high accuracy loss; for example, the use of a 16-bit fixed-point led to a 15% accuracy loss for the target application.
- The number of used BRAMs is high since we are not using any external DRAM for memory read/write operations. The range of the number of LUT and DSPs is expected due to the level of parallelism introduced using HLS directives. For the target FPGA, this is not a problem as long as we obtained a relatively reduced time latency and power consumption in the case of Approximate TSVM.
- Using "cross-layer" approximate computing: with an accuracy degradation of 4%, the approximate TSVM requires about 43% less hardware resources and classifies an unseen sample 2.64× faster while consuming 50% less power compared to its exact counterpart.
- The accuracy loss due to the use of "cross-layer" approximate computing is not the sum of the losses obtained for each single approximate technique. This is evident in the final results presented in Table 3.8.
- The use of ACTs shows a remarkable reduction in the energy per classification up to 82%, since such techniques affect both the time latency and power consumption of the TSVM, as shown in Table 3.8.
- Applying the adopted HLS optimization directives offered an additional speedup gain to the approximate TSVM in terms of speedup up to 3.2× accompanied with 84% less energy per classification.. This added a negligible overhead less than 1% increase in the hardware resources and power consumption. This is expected due to the fact that pipelining offers a reduction in the number of clock cycles while increasing the resources/power consumption. However, such increase is compensated by the dataflow directive that allows resource sharing, providing an enhanced overall implementation.

3.6 Conclusion

An approach for applying algorithmic approximate computing techniques on machine learning algorithms has been introduced. The approach has been validation on k-Nearest Neighbour and Support Vector Machine algorithms for tensor-based tactile data. A detailed study is

performed on the effect of algorithmic level ACTs on the execution time and memory usage of a kNN classifier. Results showed that an approximate kNN with cross-layer techniques could achieve reductions up to 60% with an accuracy loss less than 10%. Based on such results, the FPGA implementation of both approximate kNN and TSVM is presented. The implementation results have validated the efficiency of using algorithmic level ACTs for accelerating machine learning algorithms. Hence, the implementations could be presented as a solution for embedding intelligence on resource-limited device (e.g. Zynqberry) and power-constrained applications such as prosthetic [141].

Chapter 4

Efficient Selection-Based K-Nearest Neighbor Architecture on Modern SoCs

4.1 Introduction

Modern System-on-Chips (SoCs) are designed with heterogeneous architectures to support a variety of computationally intensive tasks in many application domains such as IoT systems, industrial automation, robotics, etc. These systems could consist of multi-core processors or Multi-Processor System On Chip (MPSoC), which could be implemented on Graphics Processing Units (GPUs), Application Specific Integrated Circuits (ASIC), or Field Programmable Gate Arrays (FPGAs). The latter is used for accelerating complex operations and performing tasks concurrently compared to traditional processors.

K-Nearest Neighbor (kNN) is a supervised classification algorithm used in a variety of applications such as pattern recognition, computer vision and machine learning [142]. However, kNN imposes significant computational workload since the complexity increases linearly with the size of the dataset and the number of classes [143]. Such workload demands significant memory requirements with high latency and power consumption [144]. Accordingly, the implementation of kNN on embedded systems with limited available energy and resources introduces a design challenge, which makes kNN hardware acceleration a necessity.

kNN algorithm involves independent operations e.g. the distance computation between a point A and point B is independent of that between points A and C. Thus, kNN doesn't require the sorting of the entire distance vector to find the K-Nearest Neighbors. Such characteristics could be exploited to reduce the computational complexity of the algorithm using a pipelined architecture and tweaking the sorting process.

In this chapter, we propose the design and implementation of a kNN architecture that is characterized by a novel selection-based sorter (Selector). The proposed selector overcomes similar state of the art solutions by reducing the occupied hardware area by up to 48% while providing a speedup up to 4.5×. The proposed kNN architecture is implemented using both exact and approximate computations. The approximate architecture utilizes the use of algorithmic level Approximate Computing Techniques (ACTs). When validated on a touch modality classification problem, both the proposed exact and approximate kNNs offer a real-time classification while consuming 6 μJ and 1.9 μJ respectively when implemented on Xilinx Zyqnberry platform. Compared to similar kNN architectures, the proposed kNN achieves a speedup between 1.4× and 875× with 41% to 94% less energy consumption and 12% to 94% average hardware area reduction. Moreover, applying algorithmic level ACTs on the proposed architecture improves its performance by achieving a 56.4% average area reduction, a speedup by 2.3×, and an energy reduction of about 69%. An accuracy degradation of 2.6% has been reported using the proposed approximate architecture. For the rest of the chapter, the term *"performance"* is used to report the characteristics of a kNN hardware implementation in terms of area, time latency, power consumption, and energy per classification. While, the term *"quality"* reflects the highest classification accuracy that a kNN architecture could achieve.

4.2 Proposed k-NN Hardware Architecture

4.2.1 k-Nearest Neighbor Algorithm Overview

The kNN algorithm classifies an input sample according to the class of the majority of K-nearest samples. For an input sample, the kNN classifier:

1. Calculates the distance between the input sample and all the samples in the training set $T_i \in T$.
2. Sorts the distances in ascending order.
3. Selects an output class based on the minimum distance towards K neighbors.

4.2.2 Selection-based kNN Architecture

The block diagram of the proposed hardware architecture is shown in Figure 4.1. The “kNN Classifier” block has been designed in HLS (coded in C++), whereas the other blocks are existing Intellectual Property (IP) blocks embedded in Vivado. The SDRAM memory is

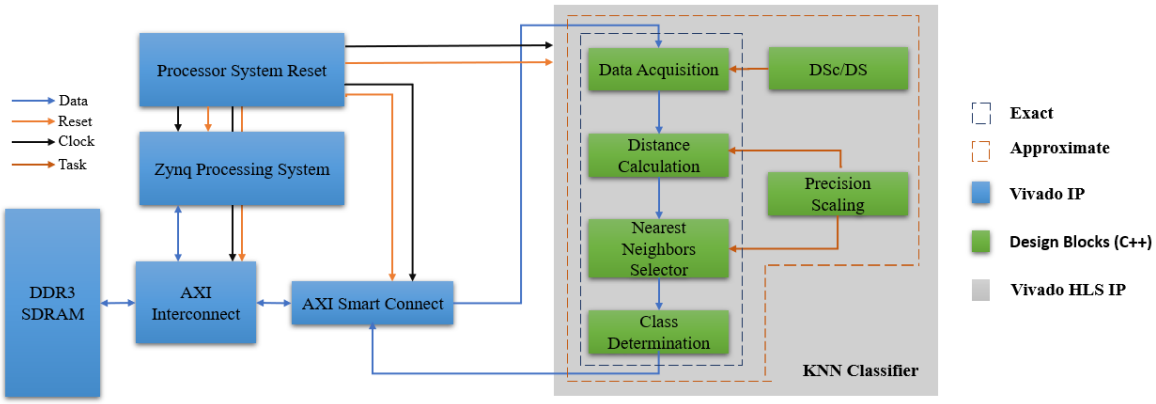


Fig. 4.1 Selection-Based kNN Hardware Architecture

used to store the training set, which is more suitable than the Block RAM (BRAM) of the FPGA for platforms with limited number of BRAMs or applications with large datasets. The Advanced eXtensible Interface (AXI) Interconnect IP handles the read and write operations from and to the memory. It adopts an AXI smart connect IP to use one AXI port for 1) writing to the data acquisition block and 2) reading the classification result from the class determination block. The Zynq processing system IP is the main block of the design which uses a processor system reset IP to drive all blocks with a common clock and reset signals. The kNN HLS IP starts operating once the Data acquisition block receives the training samples. To reduce the access overhead imposed by DRAM, we fetch the samples in bursts to reduce the number of memory accesses. Such technique has shown its efficiency in [135] and [44].

4.2.3 Nearest Neighbors Selector

A common characteristic of the most existing efficient kNN architectures is the use of the conventional sorting algorithms without optimizations for the kNN algorithm. The efficiency of these algorithms is affected by the: i) need to sort all the vector's elements, ii) large number of required comparators, and iii) increased complexity/latency for large vector size. In this work, the proposed kNN architecture avoids such sorting algorithms and adopts a selection-based one with a re-configurable division ratio for complexity and latency trade-offs.

The main idea of the "Selector" is to find the K-minimum distances without sorting the entire vector [145] as depicted in Algorithm 1. While coding the selector in HLS, the minimum K-distance values are saved in the same vector to be sorted, thus decreasing the memory footprint. The selector operations can be detailed in three steps:

Algorithm 1: Nearest Neighbors Selector

Input: Vector V with size S, Division ratio a:b, Number of neighbors K
Output: V with the K-minimum elements at the first K indices

```
S1 ← a × S/100
for i ←K to S1 do
    if V[i] <= V[0] then
        V[K−1] ← V[K−2]
        ⋮
        V[0] ← V[i]
    else if V[i] <= V[1] then
        V[K−1] ← V[K−2]
        ⋮
        V[1] ← V[i]
    ⋮
    else if V[i] <= V[K−1] then
        V[K−1] ← V[i]
for j ←S1 to S do
    if V[j] >= V[K−1] then
        break
    else
        if V[j] < V[K−2] then
            ⋮
            if V[j] < V[0] then
                V[0] ← V[j]
            else
                V[1] ← V[j]
        else
            break
```

- Step 1: The distance vector V of size S is divided into two vectors $V1$ and $V2$. The suitable division ratio ($a\%$:$b\%$) is determined via a software simulation. Begin by decreasing the size of V until the classification accuracy drops to obtain the value of a. Hence, $b = S - a$.

- Step 2: K-registers are initialized with a maximum value (e.g. 1000). Each distance value $V1[i]$ is compared to the content of register 1. If it is smaller, $V1[i]$ occupies the register, and the old content in register 1 is shifted to register 2. Then, the content in register 2 is shifted to occupy register 3. Consequently, the content in register i is shifted to occupy register $i+1$. Else, $V1[i]$ is compared to the next register, and so on.

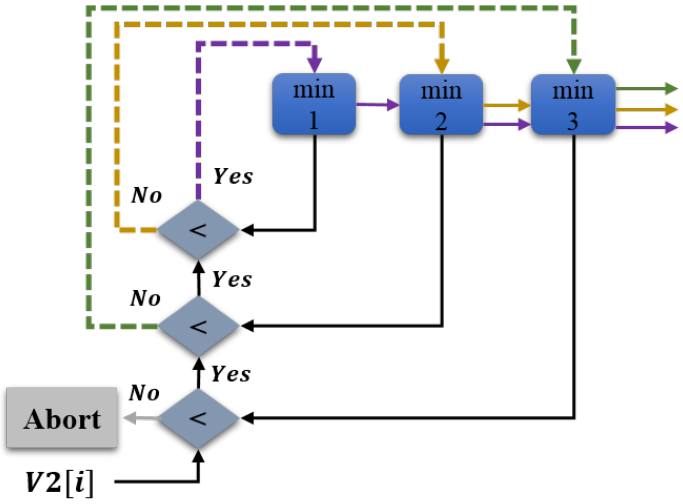


Fig. 4.2 Sorting Process Step 3 (K=3): Dashed Line (new value), Solid Line (old value), Colored Lines (concurrent operations)

At the end of step 2, the K minimum distance values are saved in the K registers in the order $min1 < min2 < ..... < minK$.

- Step 3: Each distance value $V2[i]$ is compared to the highest minimum i.e. $minK$ as shown in Figure. 4.2 (K=3). If it is larger, the minimums obtained from $V1[i]$ are not updated and a new value of $V2$ is fetched. Else, $V2[i]$ is compared to the other minimums to reach a register to occupy. Once $V2[i]$ occupies a register, the old value of that register is shifted to occupy the register of the next minimum.

The advantage of this architecture compared to the one presented in [146] is that step 3 will not be executed if $V2[i]$ is greater than $minK$. Thus, the K minimum distances are the output of step 2. This will result in a reduced selection time for hardware implementations. Moreover, the architecture in [146] selects the K minimum distances in a single step, which imposes hardware complexity and increased time latency for large datasets. While in the proposed architecture, the selection is performed in smaller steps with a high probability that the final step will not be executed ($V2[i] > minK$).
Although the proposed selector finds the K-nearest neighbors without sorting the entire vector, a comparison with two sorters reported in the literature, i.e. QuickSort [47] and Bitonic Sorter [147], has been carried out. In general:

- Bitonic sorting is a recursive algorithm that sorts a Bitonic sequence in a parallel operating fashion. A Bitonic sequence is a sequence of M elements in which L elements out of M are sorted in ascending form, and the other $M-L$ elements are

sorted in descending order [148]. If the sequence is not Bitonic, an additional task is imposed before the ability to sort the vector. The proposed selector can operate on any vector form.

- QuickSort selects one of the elements in the sequence to be the pivot and divides the sequence into two sequences. One sequence contains all the elements less than the pivot while the other contains all the elements greater than the pivot. The process is recursively (add more burden on the hardware) applied to each of the sub sequences. QuickSort has a worst-case complexity of $O(n^2)$ when the given sequence is sorted; this resembles the best-case scenario for the proposed selector as it will select the K minimums faster (the minimums occupy the first K registers). Then, all the comparisons fail. Thus no shift operations are performed. Consequently, step 3 is not executed at all.

- The number of comparators required by the selector depends on the number of neighbors K, while it depends on the size of the vector N in the case of Bitonic and QuickSort. In machine learning applications, usually, it is valid that K (the number of Nearest Neighbors in kNN) $\ll$ N (size of training vector). Given that the selector doesn't sort the complete vector, the number of comparisons is decreased.

Both the sorters presented in [47], [147], and the selector have been coded in C++ using Vivado HLS. The distance vector V has been used as a testing vector for the three implementations. In this work, the best K value and division ratio were determined to be 3 and 6:4 (for the case study presented in section 4.3) i.e. there is a need to sort only 60% of the vector and step 3 can be aborted without affecting the selection process accuracy. The obtained synthesis results for finding the three minimum numbers in V are presented in Table 4.1. The obtained results show that the selector occupies less hardware area than the implementations of both sorters. Specifically, an average reduction of 21.4% and 48.7% is reported compared to the sorters in [47] and [147] respectively. Concerning the time latency of the sorting process, the selector is faster than the sorter in [147] by 4.5×. Compared to the sorter in [47] that adopts one of the fastest sorting algorithm (QuickSort), the *performance* depends on the selection of the division ratio in step 1 and the number of neighbors K. In fact, if all the minimum numbers are located in $V2$, or if the required value of K is very large, the selector is now sorting all the elements of V resulting in a slower sorting process. Hence, a speedup of $\pm$1.2× (for 6:4 and 5:5 ratios respectively) has been observed for the given task.

Table 4.1 HLS Synthesis Results of Different Sorters

Sorter	FF	LUT	Clock Cycles
Proposed Selector	84	176	15 (division ratio 6:4)
[47]	106	226	17
[147]	123	514	96

4.2.4 Approximate kNN Blocks

In chapter 3, we have presented a complete assessment of using algorithmic level ACTs on a kNN classifier. The assessment included the degradation in accuracy to the gain in memory and execution time on Intel i7 CPU. The studied techniques have been formulated into a general approach that has been tested on two machine learning classifiers. The reported approach has been adopted in the proposed kNN architecture where a trade-off between the classifiers *performance* and *quality* has been considered. The trade-off resulted in our selection for the ACTs presented in the proposed approximate kNN architecture. All the adopted techniques belong to the data-oriented approximate computing category [149]. The adopted ACTs are Dataset Reduction (Downsampling (DS) and Downscaling (DSc)), and Data Format Modification (DFM). DS means varying the signal sampling frequency during signal acquisition. Since tactile data used in this work are from an already available dataset, the sampling frequency can't be changed. As a consequence, DS is applied offline on the dataset by reducing the number of samples. DSc is applied by adjusting the sample size as shown in section 4.3.1. As for DFM, fixed-point representation is adopted, and the precision is determined as a trade-off between resolution (32, 24, 16, and 8-bit) and classification accuracy." The approximate kNN classifier starts operating once the Data Acquisition block receives the training samples (after DS/DSc has been applied offline) from the memory. Then, the same steps performed by the kNN HLS IP are executed.

4.3 Case Study: Tactile Data Processing for Electronic Skin Systems

4.3.1 Electronic Skin Overview

Electronic skin (E-skin) system is an artificial system developed to mimic human skin behaviour or to implement intelligent tasks in applications such as robotics, prosthetic, etc. The E-skin is composed of a set of components as shown in Figure 4.3. At the moment of touch, the array of tactile sensors transforms the applied mechanical stimuli into electrical

signals. The electronics interface is in charge of data acquisition, signal conditioning, and analog to digital conversion. After that, the tactile data should be processed by the embedded data processing unit. To achieve this goal, various processing levels may be applied. For example, simple processing algorithms could be employed to retrieve information such as direction and intensity of contact force, contact location, temperature, etc. On the other hand, more sophisticated algorithms should be adopted when targeting complex/smart processing tasks like textures, patterns, and objects recognition, or touch modalities, and roughness classification [150], [151]. To this end, the smart embedded data processing unit implements ML algorithms in order to enable the above-mentioned smart tasks. However, implementing ML algorithms on hardware platforms is challenging due to the high complexity of such algorithms. Consequently, affecting the complexity of the embedded electronic systems in terms of time latency and power consumption.

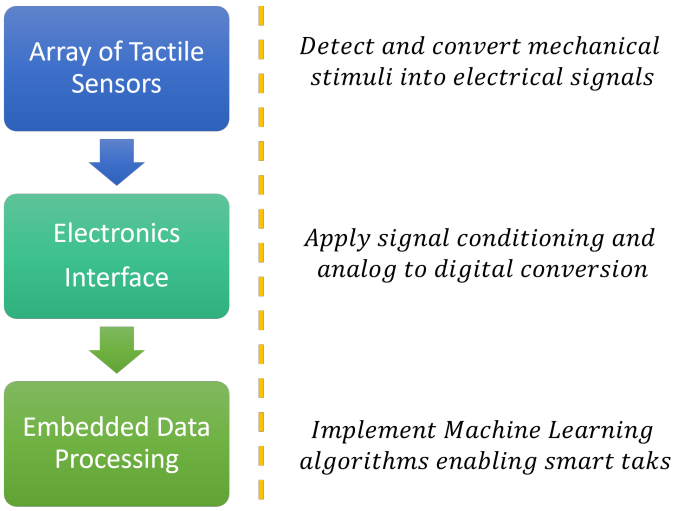


Fig. 4.3 Electronic skin system and the corresponding function of each block

In this work, the kNN algorithm is adopted for the design of an embedded tactile data processing architecture due to the: 1) high level of parallelization of the kNN algorithm, which makes it adequate for hardware acceleration, 2) high classification accuracy with a reduced computational complexity compared to state-of-the-art algorithms operating on the same task [32], [152], and 3) ability of complexity reduction without affecting the application *quality* using approximate computing techniques as reported in chapter 3.

4.3.2 Experimental Setup

The dataset collected in [5] describing several touch modalities has been selected for the validation of the proposed kNN architecture. The experimental setup is shown in Figure 4.4 and can be described as follows:

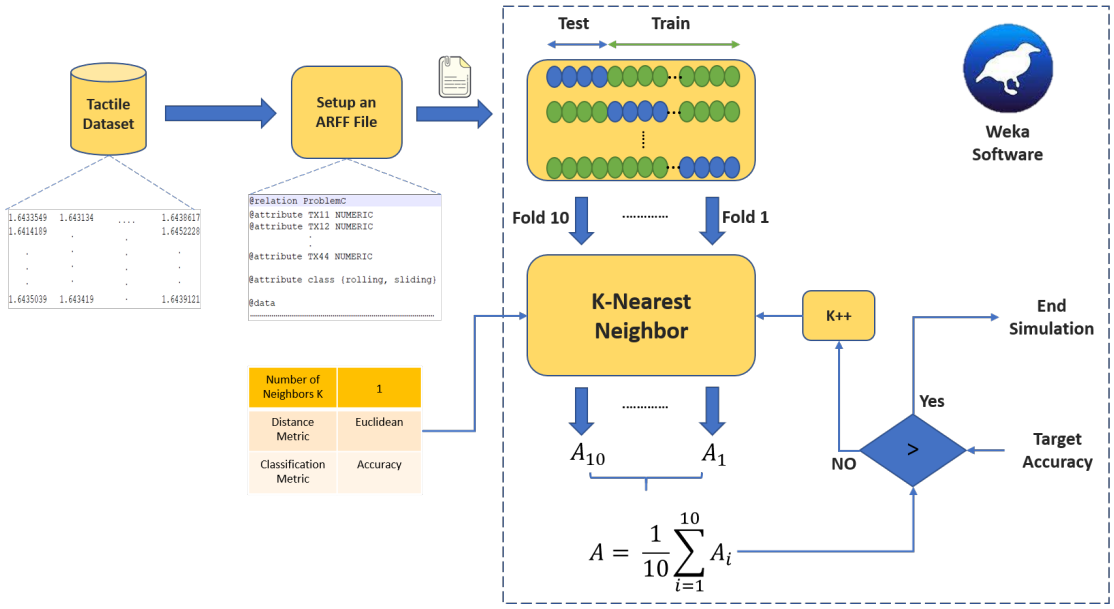


Fig. 4.4 Experimental Setup

- Dataset: The dataset contains records for the two touch modalities performed by 70 participants. Each modality was recorded from a 4×4 tactile sensor for 10 seconds at 3 kHz sampling frequency. Thus, each raw data sample can be modeled in the form of a tensor of size $4 \times 4 \times 30{,}000$. The touch modalities were performed on both the horizontal and vertical directions for two trials, resulting in a dataset of 840 samples.

- Simulation Software: An open-source machine learning simulation tool called "Weka" has been used [153]. Weka involves a collection of learning algorithms that can be applied to a pre-defined dataset or invoked from a Java code. The tool has options for classification, clustering, regression, etc.

- Classification Task: "Sliding a finger" vs "Washer Rolling" binary touch modality classification problem introduced in [5]. For Weka simulation an Attribute-Relation File Format (ARFF) file is required. Thus, a header describing the features and the possible output class of each sample is added to the original tactile dataset.

- kNN Characteristics: A 10-fold cross-validation simulation using the Weka tool has been carried to determine the best value of K. The adopted distance between two tactile samples $T1$ and $T2$ is the squared euclidean distance written as:

$$d(T_1, T_2) = \sum_{0}^{16} (TX_{ij} - TX_{mn})^2 \tag{4.1}$$

  where TX_{ij} and TX_{mn} are the taxels inside a 4×4 tactile sample.

- Classification Metric: The kNN algorithm has been assessed by calculating the classification accuracy i.e. the ratio of correctly classified samples to the total number of available samples.

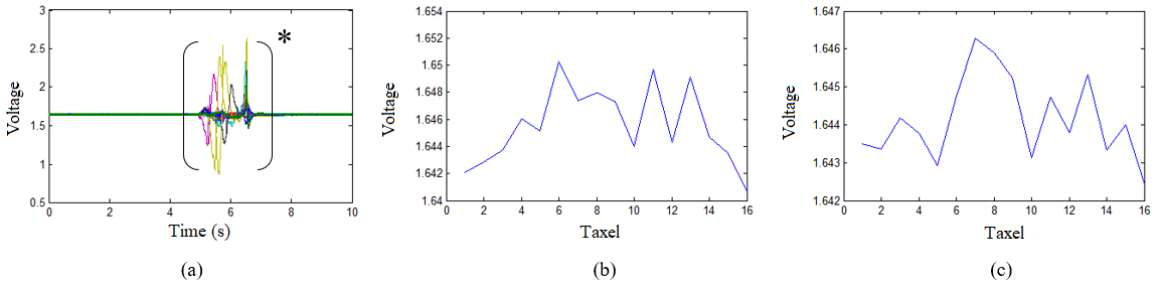
The ARFF file was loaded into Weka and classification using kNN is performed. A kNN classifier with 3-nearest neighbors resulted in the highest classification accuracy of 89.8%. This result was achieved based on a model selection approach.

The best obtained kNN model with K=3 is referred to as Exact kNN. As for Approximate kNN, it employs the following techniques to the Exact architecture: 1) Downsampling which applies an approximate window on the touch modality, where only the data that corresponds to the interval [a,b] seconds is considered. First, the interval [a, b] is selected such that $0.1 < a \ll 1$ and $9.5 < b < 10$. Then, the values of a and b are varied, while calculating the classification accuracy. The interval [3.5, 7] provided the highest accuracy among others. Thus, each modality tensor can be written as $\phi = 4 \times 4 \times 10,500$ (touch readings that belong to the interval [3.5, 7] seconds), and 2) downscaling which reduces the tensor size to $\phi = 4 \times 4 \times 1$, where the last dimension is the mean of the 10,500 readings according to the equation:

$$mean = \frac{\sum TX_{ij}}{10500} \tag{4.2}$$

where TX_{ij} is the individual taxel inside the 4×4 tactile sample. Figure 4.5 shows the initial and obtained touch modalities after applying DS and DSc.

It is worth mentioning that the tensor representation of data has been adopted by [5] since it preserves the initial structure of the data, which is still valid after applying approximate computing techniques. This is evident in Figure 4.5 (b), (c) where the two touch modalities can still be differentiated. For both the Distance Calculation and Nearest Neighbor Selection blocks, the operations are implemented in 24-bit fixed-point representation with a $< 6, 18 >$ precision. The adopted precision is based on a trade-off between complexity and classification accuracy.



(a) (b) (c)

Fig. 4.5 Touch Modalities: (a) Rolling with DS, (b) Rolling after DSc, (c) Sliding after DSc, * Window

4.4 Selection-based kNN Implementation

4.4.1 Hardware and Software Design Tools

The Zynqberry TE0726-03M [154] has been adopted for implementation. Zynqberry is a small-sized platform in the form of a Raspberry Pi compatible System-on-Chip (SoC) module integrating a Xilinx Zynq-7010 with a 512 MB Synchronous Dynamic Random Access Memory (SDRAM) memory. The Zynq SoC has a hybrid structure of combining a dual-core ARM Cortex-A9 processor as a Processing System (PS) and an FPGA as Programmable Logic (PL) in a single SoC. Moreover, for comparison purposes, the architecture has been also implemented on the Virtex-7 FPGA and the NVIDIA GTX 1650 GPU.

As for the software tools, Vivado HLS 2018.3 and Vivado 2018.3 were used. Vivado HLS allows the design of an embedded system on FPGA using a high-level programming language such as C and C++ compared to traditional hardware description languages (HDL). The use of HLS decreases the FPGA development time and effort. Also, it offers a set of optimization directives that can be used to enhance the design *performance*.

4.4.2 Implementation Methodology

Once the whole design code is completed in HLS and design optimizations are applied, a co-simulation is performed. This simulation runs both the C++ and the RTL simulations together to verify a matching output. Then, the design is exported as a Register Transfer Level (RTL) IP block. The latter is imported into Vivado 2018 and connected to the Zynq processor and other IP blocks as seen in Figure 4.1. First, a behavioral simulation is performed to verify the functionality of the design. Then, synthesis and place and route occur to finalize implementation. At this point, the generated report contains the occupied area percentage (BRAM, DSPs, etc.) and the number of clock cycles passed to generate an output.

Concerning power consumption estimation, Vivado offers two methods: Vector-based and Vector less. This estimation can be performed at any stage between post-synthesis to post routing. For a credible estimation, a post-implementation functional and timing simulation is used to generate a Switching Activity Interchange File (SAIF) to be used for a vector-based estimation post-routing.

4.4.3 Design Optimization

Targeting the real-time functionality on a small-sized platform such as Zynqberry, the proposed architecture has been optimized to ensure an acceptable balance between time latency and hardware requirements. This has been achieved with several design optimizations as shown in Figure. 4.6. Such optimizations are facilitated with the use of Vivado HLS directives [140] as depicted in Algorithm 2. These optimizations are summarized as follows:

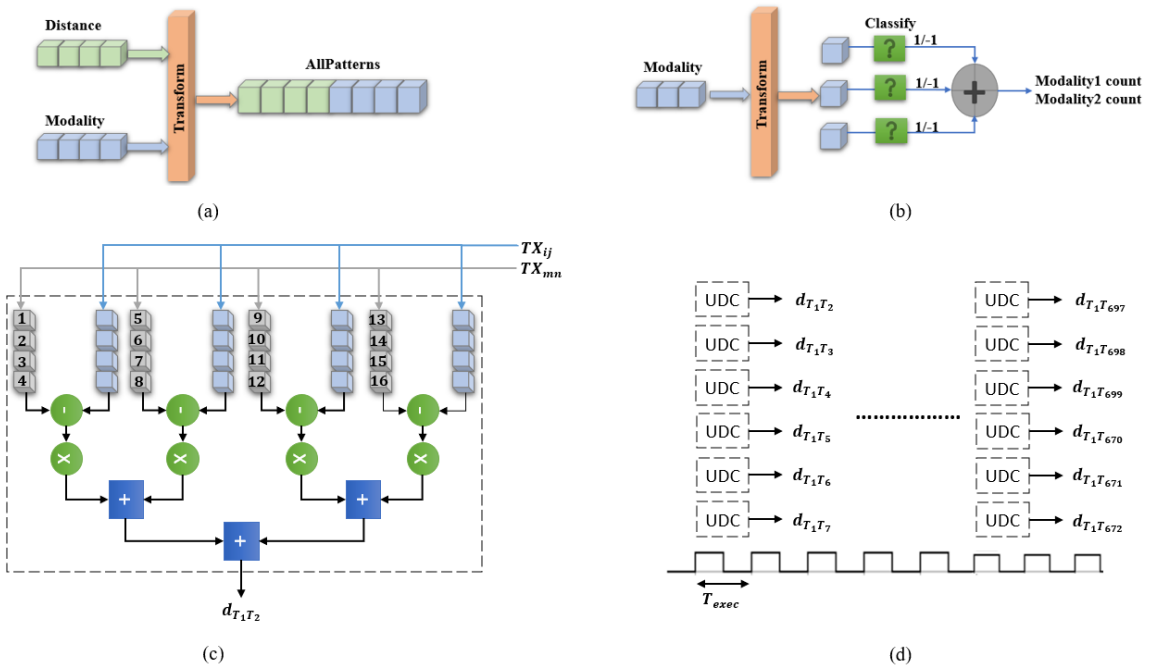


Fig. 4.6 Design Optimization: (a) BRAM Resources Reductions, (b) Unrolled Class Determination, (c) One Unrolled Distance Calculation (UDC) Block, (d) Complete Unrolled Distance Calculation

- BRAM Resources Reduction: As the BRAM size is 18KB in the FPGA, it is better to combine many arrays into a single array if their sizes are less than 18KB. Since kNN is a supervised algorithm, the class of each query must be known. Thus, we can benefit from this directive to combine the “Distance” and “Modality” arrays into a single array

as shown in Figure 4.6(a). Thus, when the selector block finds the three minimum neighbors, the class of each selected neighbor is available at the same instant. This process is referred to as "Array Map" where the horizontal option means that the two arrays are combined into a single array with more elements (see Algorithm 2).

Algorithm 2: kNN Design Optimization

```
#pragma HLS ARRAYMAP
variable=Distance Instance=AllPatterns horizontal
variable=Modality Instance=AllPatterns horizontal
/* M: nb of features                                          */
function UDC(T1, T2):
for i ←0 to M do
    #pragma HLS UNROLL factor=4
    Execute (4.1)
/* Tactiles:  trainingset, q:  testing point                  */
/* N: nb of training points                                   */
for i ←0 to N do
    #pragma HLS INLINE
    #pragma HLS UNROLL factor=6
    Distance[i]= UDC(p, Tactiles[i])
    Modality[i]=Tactiles[i][16]
/* K: nb of Neighbors                                         */
NearestNeighbors=Selector(K, Modality)
for i ←0 to K do
    #pragma HLS UNROLL
    if NearestNeighbors[i] == 1 then
        modality1count++
    else
        modalitycount2++
```

- Parallelization: To exploit the capabilities of the FPGA, the operations of the distance calculation and the class determination blocks are executed in parallel with a small unroll factor. In HLS terms this is known as "Unrolling". Unrolling a loop creates multiple copies of its body in the RTL design, which allows some or all of its iterations to occur in parallel. This optimization has been applied to (4.1) denoted as Unrolled Distance Calculation (UDC) and the Class Determination blocks leading to accelerating the calculation and output class decision. However, executing all the operations in parallel leads to a high power consumption and increased resource requirements. To avoid the negative impact of unrolling on the hardware cost and power consumption, the loops are partially unrolled (unroll factors of 4 and 6) as it is shown in Algorithm 2,

and the design is implemented at an operating frequency of 100 MHz which is lower when compared to similar work [147]. Each touch modality sample has 16 features, thus an UNROLL factor equals to 4 is used. This means that the distance between each 4 features is calculated in a single time interval as shown in Figure 4.6(c). Similarly, Figure 4.6(d) shows how the UDC block is used to calculate the distance between the testing sample and all training samples. An UNROLL factor equals to 6 is used, thus 112 timing intervals are required to finish all the distance calculations for a training set size of 80%. The distance from a testing sample to $(80 \times 840/100) = 672$ training samples is calculated in batches of 6 calculations per timing interval i.e $672/6 = 112$ intervals. As for Class Determination, since $K = 3$ and we have a binary classification, the process could be fully unrolled as shown in 4.6(b).

- Function Inline: The inlined function is treated as a part of the calling function that is calling it rather than a separate entity. This optimization is applied for the distance calculation function. Thus, whenever the classification function is called, the distance calculation is executed within it and it no longer appears as a separate level of hierarchy in the RTL design. Thus improving the overall latency of the classification task.

4.4.4 Implementation Results

The *performance* and *quality* of the proposed exact and approximate implementations are assessed on the touch modality classification problem mentioned in section 4.3. The assessment involves two case studies: (1) Proposed Exact kNN versus approximate kNN and (2) Exact kNN on FPGA versus GPU. For the FPGA implementation, the time latency T is calculated according to the equation:

$$T = N \times 1/f_{max} \tag{4.3}$$

where N is the number of clock cycles obtained in post-implementation reports and f_{max} is the maximum operating frequency the design can achieve. The Joule per classification energy E is calculated as:

$$E = T \times P \tag{4.4}$$

where T is the time latency and P is the dynamic power consumed by the programmable logic (PL) of the Zynqberry reported by Vivado i.e the power consumed by the simulated kNN architecture to compute a classification of an input sample (excluding the static power of the processing system (PS) as it is device dependent).

For GPU implementation, Exact kNN architecture has been coded using Python language running inside the CUDA computing platform. The GPU power estimation was obtained using NVIDIA System Managment Interface (NVSMI). The latter is a command utility that can be issued on any Python development environment with the CUDA libraries imported [155].

Case 1: Exact versus Approximate kNN

The implementation results on Zynqberry of the proposed Exact and Approximate kNN are shown in Table 4.2. Exact kNN occupies 12% of the hardware resources, consumes 0.236 W, and classifies an input sample within 25.7 μs. The obtained time latency verifies the real-time classification of a touch modality in less than 400 ms [4]. Applying down-sampling and downscaling have decreased the input size from $4 \times 4 \times 30,00$ (a 10s sample) to $4 \times 4 \times 1$ (a 3.5s sample) offering a 65% reduction in data size. Such reduction led to a significant decrease in the hardware resources and time latency. Using the 24-bit fixed-point representation instead of 32-bit floating-point one provides a 25% reduction in the word length of data exchanged between the different blocks of the kNN architecture and the complexity of the arithmetic computations. Consequently, the dynamic power consumption has been reduced. Thus, the proposed approximate kNN offers an average hardware resource reduction up to 56.4%, by accelerating the classification of a test sample by 2.3$\times$ with an energy reduction of about 69% compared to the proposed Exact kNN. For the whole design, an accuracy degradation of 2.6% is reported. The proposed approximate kNN provides real-time classification of touch modalities with a reduced time latency of 11.2 μs. These results motivate the use of approximate computing techniques.

Table 4.2 Implementation Results of the proposed Exact and Approximate Classifiers on Zynqberry

Implementation	Exact	Approximate
Classification Accuracy	89.8%	87%
Frequency (MHz)	100	
BRAM	4	4
DSP48E	10	4
FF	3493	1612
LUT	2825	1264
Time Latency (μs)	25.7	11.2
Dynamic Power Consumption (W)	0.236	0.164

Case 2: Exact kNN on FPGA versus GPU

Using the CUDA platform and NVSMI tool, the GPU implementation of Exact kNN provided a classification time of 80 ms while consuming 14.12W for the touch modality classification problem. Table 4.3 shows a comparison between the FPGA and GPU implementations in terms of execution time and energy consumption. The results are significantly in favor of the FPGA, where the acceleration of the proposed kNN architecture on FPGA could be achieved with a fraction of the energy consumed using the GTX 1650 GPU. This can be justified due to two possible reasons:

Table 4.3 Exact kNN performance on FPGA and GPU

Exact kNN	
Time (FPGA/GPU)	25.7μs/80ms
Energy (FPGA/GPU)	6μJ/1.13J
Time Ratio	0.00032
Energy Ratio	5.37×10^{-6}

- GPUs use DRAM for the communication between the different blocks of the kNN architecture (referred to as kernels) [156], which is slower than using a hybrid structure as proposed in Figure 4.1 where BRAMs are used to communicate between different blocks and DRAM is used only for dataset storage.
- The proposed kNN architecture exploits the parallelism capabilities of the FPGA. Thus, the "if-then-else" conditions are executed in parallel. On the other hand, the "then" and "else" parts are executed serially on GPUs resulting in a significant time latency increase. Such issue is known as "thread divergence" [157].

4.5 Comparison with existing solutions

Comparing two kNN implementations is not a straight forward task due to the large number of differences such as: the number of nearest neighbors (K), dataset size (N), number of features per sample (f), development environment (HLS or HDL), hardware device used, etc. To achieve a fair comparison with Exact kNN, three similar architectures have been selected. These three architectures have been chosen such that they all have:

- Used HLS for development since the comparison with an HDL implementation is not feasible.
- Achieved a high acceleration gain (i.e speedup) with respect to equivalent CPU-based kNN implementation, so the architecture resembles an efficient accelerator.

- Used similar (and different) values of K, N, and f to generalize the comparison.

Table 4.4 presents the existing implementation settings for the three architectures. Denote by $S_i = [K_i, N_i, f_i, Dataset_i]$ the settings used in the first [146], second [158] and third implementation [147] respectively. Exact kNN is implemented using the settings S_i i.e the proposed kNN architecture is implemented and validated using the testbench reported in each architecture. The implementation results are shown in Table 4.5 with the original implementation results of each architecture.

Table 4.4 Testbench Implementation settings for the Exact kNN and three similar architectures

Architecture	S_1 [146]	S_2 [158]	S_3 [147]
K	10	3	5
N	699	150	300×10^3
f	9	4	2
Dataset	BCW_9	Iris	Weather
Device	AVNET ZedBoard	Virtex-7	Virtex-7
Frequency (MHz)	100	100	240

Table 4.5 Proposed Exact kNN Implementation Results versus Similar Solutions

Architecture	kNN-S_1	[146]	kNN-S_2	[158]	kNN-S_3	[147]
Device	Zynqberry				Virtex-7	
Frequency (MHz)	100				240	
BRAM	4	-	4	293	500	512
DSP	7	9	5	47	12	12
FF	2002	9484	827	-	21677	23892
LUT	1607	8845	1407	-	11416	11838
Time Latency (ms)	22×10^{-3}	0.27	12×10^{-3}	10.5	0.88	1.24
Energy per classification (mJ)	4.84×10^{-3}	70×10^{-3}	-	-	1.86	3.17
Average Resources Reduction (%)*	61%		94%		12.3%	
Speedup	12.3×		875×		1.4×	
Energy Reduction (%)	94%		-		41.5%	
Classification Accuracy (%)	96.2%		93.3%		86.5%	

* Calculated for the available resources only e.g. reduction in BRAM and DSP for kNN-S_2 compared to [158], i.e. Reduction = (BRAM-Reduction + DSP-Reduction)/2, where BRAM-Reduction= 100(1-4/293) and DSP-Reduction= 100(1-5/47).

kNN-S_1 achieves a 12.3× classification speedup with 94% less energy consumption while requiring a 61% less hardware resources compared to the kNN implementation presented

in [146]. This is due to two main reasons: 1) the selector used in kNN-S_1 is an enhanced version of the one used in [146] where the division factor plays a key role in decreasing the sorting time as presented in section 4.2.3, and 2) the aim of the kNN architecture in [146] is to attain the highest speedup possible for real-time embedded applications. This has been accomplished by combining the UNROLL, PIPELINE, and DATAFLOW directives. Such directives are known for speedup gains due to the level of parallelism introduced at the expense of a noticeable increase in the hardware resources. The latter was not an issue when using the relatively large FPGA in the ZedBoard platform. Meanwhile, in kNN-S_1 only the UNROLL directive is used with an unrolling factor that balances the speedup and complexity for the target application, while achieving more speedup with the use of the selector.

When compared to the kNN implementation in [158], kNN-S_2 provides a huge acceleration gain of 875× with 94% less required hardware resources. Such gain is due to the design choices adopted by the authors in [158] such as: 1) using the euclidean distance metric, which compared to (4.1), has an added complexity due to the square root operation, 2) applying the normalization of the data on-chip, which presents a complexity overhead, and 3) performing the sorting operation using a single comparator and multiple BRAMs to compare each pair of data points, this process is very slow compared to the proposed selector. Although no power/energy details were provided for the kNN in [158], kNN-S_2 is expected to be more efficient due to the 90% reduction in the number of DSPs.

The implementation requirements of kNN-S_3 exceeds the capacity of the FPGA fabric in Zynqberry and thus the design couldn't be routed to achieve the 240MHz operating frequency. Thus, for comparison reasons only, kNN-S_3 is implemented on the target device used in [147] i.e Vertix-7 knowing that implementing kNN on Zynqberry has achieved the real-time and low power consumption demands for the target touch modality application as reported in Table 4.2. kNN-S_3 offers a speedup of 1.4× with 41.5% and 12.3% reduction in hardware resources and energy per classification respectively. Such results are justified with the lower number of FF and LUTs required by kNN-S_3. This is expected since the kNN in [147] uses the Bitonic sorter, which is outperformed by the proposed selector as shown in Table 4.1. Compared to kNN-S_1 and kNN-S_2, the gain achieved by kNN-S_3 is relatively lower since the kNN in [147] exploits the high optimization capabilities of OpenCL for extensive computations and large datasets.

4.6 Conclusion

This chapter introduces an efficient novel architecture for the hardware acceleration of the k-Nearest Neighbor algorithm using a selection-based sorter. The architecture has been coded

in HLS, synthesized, and routed on the Zynqberry platform. Two efficient implementations have been provided based on exact and approximate computing. The implementations exploit the parallelism nature of the kNN algorithm along with the use of ACTs to achieve real-time classification with relatively low power consumption. Compared to similar state-of-the-art solutions, the proposed Exact kNN offers acceleration gain between 1.4× and 875× with lower energy per classification between 41% and 94% depending on the used settings. Compared to GPU-based implementations, the proposed kNN-FPGA implementation offers efficient and faster classification for the target application. Such results pave the way towards embedding intelligence using a small-sized platform such as the Zynqberry for applications with low power and real-time requirements. The implementation on different hardware platforms, under different settings, and operating on different dataset size, verifies the efficiency of the proposed selection-based kNN architecture. Future work will involve the investigation of the use of circuit-level approximate computing techniques that are reported to permit noticeable gains in the *performance* of machine learning hardware implementations.

Chapter 5

Real-time Accelerated Tensorial Support Vector Machine Architecture

5.1 Introduction

Tensor based learning techniques permit the effective exploitation of the structure of data used in various fields such as vision (e.g. image recognition), neuroscience (e.g. MRI data), chemistry (excitation-emission data), etc. At the beginning of the last decade, Machine Learning (ML) communities started showing interest in tensors and their use for supervised learning [159]. Authors in [54] proposed a tensorial kernel that could be used for supervised tensor-based learning models while utilizing the structural information embodied in the data and exploiting the algebraic properties of tensors of any order. Such kernel methods lead to flexible nonlinear models that have been proven successful in many different contexts. When used with Support Vector Machine (SVM) algorithm, the tensorial kernel achieved better classification accuracy than the Gaussian-Radial Basis Function (RBF) and linear kernels in an image recognition task.

Gastaldo *et. al* have extended the tensorial kernel approach for tactile data processing in [29]. This approach has been adopted for a touch modality classification problem since it preserves the inherent tensorial structure of the data collected by tactile sensors. As an end result, the tensorial-based SVM has achieved higher accuracy in classifying touch modalities compared to the Regularized Least Square (RLS) algorithm. In [73], the first FPGA implementation of the SVM algorithm based on tensorial kernel has been presented. Specifically, two implementations were provided: Cascaded and Parallel. The former failed to ensure real-time classification of touch (i.e in less than $400ms$ [4]), and the latter reported

a relatively large hardware area and high power consumption of 1.14W. Such results were not acceptable for applications with limited power budget and area constraints [141].

In this chapter, we present a new architecture and hardware implementation of the tensorial SVM (TSVM) aiming at reducing the hardware complexity and power consumption while keeping real-time operation. The architecture is characterized by the introduction of a Shallow Neural Network (NN) for the Singular Value Decomposition (SVD) computations. The proposed neural network architecture achieves 324× speedup with 58% and 67% reductions in the required hardware resources and power consumption respectively compared to the traditional one-sided Jacobi algorithm. Such reductions demonstrate the feasibility of the implemented TSVM for real-time tactile data classification while consuming 6.28 mJ. The proposed TSVM architecture achieves 131× classification speedup with a 39% and 50% resources and power reductions respectively compared to similar stat-of-the-art solution [73]. Furthermore, a scalability assessment of the proposed TSVM architecture is provided. The assessment shows that replacing the one-sided Jacobi with a neural network demands only 1% increase in the required FFs compared to 29% when the number of training tensors is doubled.

5.2 SVM Classification based on Tensorial Kernel

5.2.1 TSVM Re-visited

SVM classification of an input tensor has been introduced in a framework that extends kernel methods to tensor data in 4 main steps [5]: Tensor Unfolding, SVD Computation, Kernel Computation, and Classification (for details refer to chapter 3, section 3.5.1).

5.2.2 Complexity Assessment of TSVM

Figure 5.1 illustrates the estimated number of operations required in each step of the tensorial SVM algorithm, where m and n are the dimensions of the unfolded matrix, N_c, N_t, and N_{sv} are the number of classes to be discriminated, the number of training tensors, and the number of support vectors, respectively. As reported in [73] (m=8, n=20, N_c=2, N_t=100, etc.), the SVD computation corresponds to about 96% of the overall algorithm. In [73], the one-sided Jacobi algorithm has been adopted for finding the singular vectors. Such algorithm involves a high number of arithmetic operations and requires several iterations to converge [160]. Hence, our goal is to find an efficient alternative for the SVD computation of an input tensor.

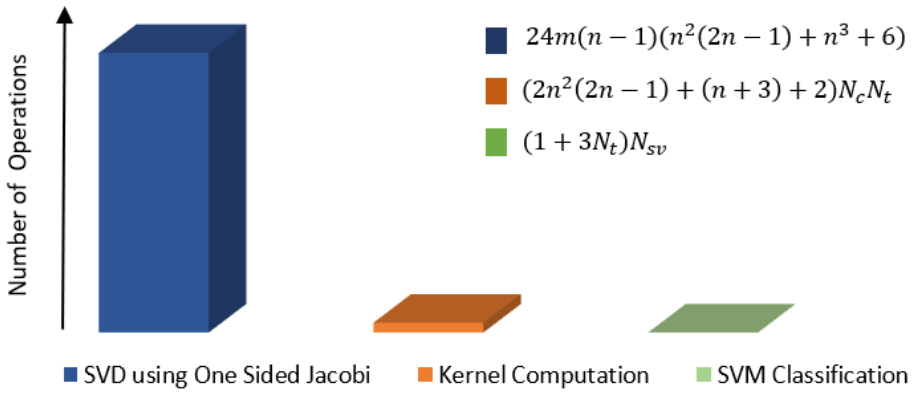


Fig. 5.1 Computational Complexity of the Tensorial SVM algorithm

5.2.3 Touch Modalities Classification

The tensorial SVM has been initially presented as an effective algorithm for touch modality classification in [5]. In this work, three binary and one multi-class classification problems are used to for the validation of the proposed TSVM architecture. Specifically, the problems are:

- Problem A: "brushing a paintbrush" versus "rolling a washer"
- Problem B: "brushing a paintbrush" versus "sliding the finger"
- Problem C: "sliding the finger" versus "rolling a washer"
- Problem D: "one versus the others"

These modalities have been derived from a tactile dataset that has been collected by 70 participants. Each participant performed the modality on both the horizontal and vertical axes of a 4×4 tactile sensor for a duration of 10 seconds and a sampling frequency of 3kHz. Thus, each touch modality could be represented by a tensor $\phi(4 \times 4 \times 30,000)$.

5.3 SVD Algorithms and Implementations

5.3.1 Literature Review

Singular value decomposition can be computed numerically through several methods such as the Jacobi method, the QR method, and the one-sided Hestenes method [161]. For parallel implementations, computing the SVD using the Jacobi method is superior to other methods in terms of complexity and execution time [161]. Brent *et. al* have shown that two-dimensional systolic arrays could be used for implementing the Jacobi method [162]. In [163], the authors

have presented various realization for the Jacobi SVD computation using Coordinate Rotation Digital Computer (CORDIC) [164]. The latter is adopted in majority of the existing hardware implementations of the Jacobi SVD method. For small matrix dimensions, an efficient implementation of SVD for the use in Multiple Input Multiple Output (MIMO) precoding and real-time signal processing has been presented in [165]. The implementation is based on CORDIC processors. For an arbitrary $m \times n$ matrix, Ibrahim *et. al* have presented an FPGA implementation with fixed-point arithmetic [166]. The implementation managed to compute the SVD of an 32×127 matrix in 13 ms while occupying 20% and 67% slice registers and LUTs respectively on a Virtex-6 FPGA. Fast and efficient FPGA implementation for computing the singular and eigen value decomposition based on a simplified CORDIC-like algorithm is presented in [167]. The implementation used fixed-point arithmetic for sequential and parallel operations leading about $3\times$ faster computation in an image denoising application compared to computations via an Intel CPU based PC. The authors in [168] used High-Level Synthesis (HLS) to model the one-sided Jacobi SVD computation on a Zedboard development board. For a 16×16 matrix, SVD computation takes around 1.1 seconds with a power consumption of 1.38W. Using CMOS 28-nm technology, Deng et.al proposed a hardware architecture for tensor SVD [169]. Compared with real-world CPU-based implementations, the architecture provides an average of $14\times$ speed on various workloads. These alternative implementations for SVD computation share several common challenges: (1) they operate only on square matrices. Thus, if the input matrix is rectangular, an additional complexity is added due to matrix symmetrization [169]. (2) if the implementation uses floating-point representation, the complexity is relatively high even for small matrix dimensions [170], and (3) depending on the required output precision, the algorithm might require additional iterations to converge [160].

A neural network is one of the candidates for the SVD computation. The idea first surfaced in 1991 when Samardzija *et. al* proposed an artificial continuous-time neural network to estimate the eigenvectors and eigenvalues [171]. In [172], the convergence and computational complexity through computer simulations of such network are assessed. Another neural network has been presented in [173]. The network is characterized by an order n-Ordinary Differential Equations (ODEs) leading to reduced dimensionality. Such neural network has evolved to further applications such as Principle Component Analysis (PCA) [174].

In this work, a new architecture for SVD computation based on shallow neural networks is proposed. The architecture offers the ability to operate on rectangular matrices (thus symmetrization is not needed (see Figure 5.2), and utilizes floating-point arithmetic. As for convergence, the neural network training is usually performed offline on a high-end

computing device. Thus, a network could be trained several times for any given amount of time to achieve top notch performance.

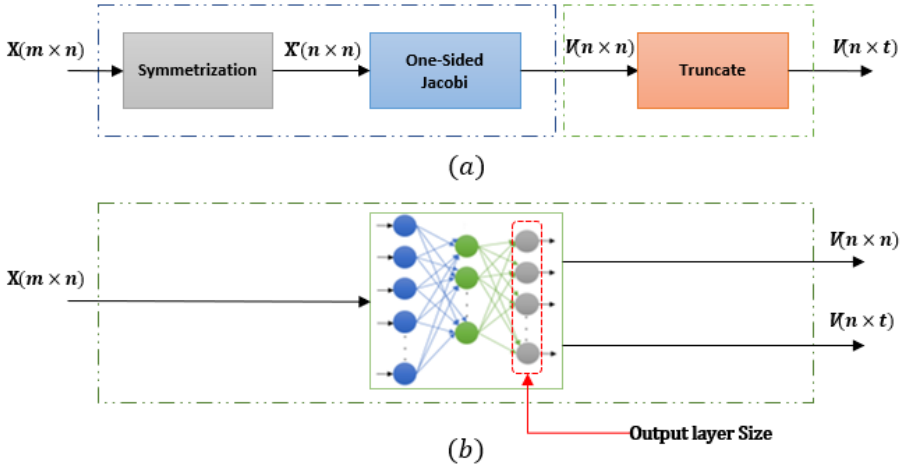


Fig. 5.2 SVD Computation using: (a) one-sided Jacobi, (b) Neural Network

5.3.2 Computational Complexity

In this section, we compare the complexity of the one-sided Jacobi algorithm with that of a shallow neural network in terms of the total number of operations. Consider a shallow neural network of one hidden layer of size H and an output layer of size O. For an input $A_{m\times n}$, the outputs of the hidden layer Y_h and the output layer Y_O are expressed respectively as:

$$Y_h = f_h(W_h.A + b_h) \tag{5.1}$$

$$Y_O = f_O(W_O.Y_h + b_O) \tag{5.2}$$

where W, b, and f represent the weight, bias, and activation function respectively. The output of each layer consists of matrix multiplication, addition, and activation operations. The number of operations for matrix multiplication and addition is expressed as:

$$N_h = H(2m \times n - 1) + H = 2H(m \times n) \tag{5.3}$$

Assuming that the activation function requires N_{Act} operations, the total number of operations in the hidden layer is expressed as:

$$N_h = 2H(m \times n) + N_{Acth} \tag{5.4}$$

The same can be applied to the output layer, thus the number of required operations is:

$$N_O = 2H \times O + N_{ActO} \tag{5.5}$$

Finally, the number of operations for the whole network could be expressed as:

$$N = N_h + N_O = 2H(m \times n + O) + N_{Acth} + N_{ActO} \tag{5.6}$$

To estimate N, suppose there exists an upper bound T such that $N \leq T$. T is an upper bound when both N_{Acth} and N_{ActO} correspond to the most complex activation function i.e. the tangent hyperbolic function (tanh). The latter is expressed as:

$$f(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}} \tag{5.7}$$

To find the number of operations required for the term e^z, we referred to the function implementation in the IEEE-754 library in [175]. The implementation uses the Taylor expansion with an order 3 for floating-points, leading to a total of 16 operations. Thus the number of operations $N_{Acth} = 35H$ (1 Add, 1 subtract, 1 divide, 16 for e^z and 16 for e^{-z} for each neuron). Similarly, $N_{ActO} = 35O$. For the network to output the right singular vectors $V_{n\times n}$ of an $m \times n$ matrix, the output layer size O is equal to n^2. This simplifies (5.6) to:

$$N <= (2H \times n)(m+n) + 35H + 35n^2 \tag{5.8}$$

Knowing that the number of operations for the one-sided Jacobi algorithm is (see Figure 5.1):

$$Nj = 24m(n-1)[n^2(2n-1) + n^3 + 6] \tag{5.9}$$

through simulations, the values of m, n, and H are varied to compare (5.8) and (5.9). Figure 5.3 plots the number of operations N_j and N required to compute the SVD of a matrix using one-sided Jacobi and a shallow neural network respectively.

Generally, the comparison results are in favor of the neural network approach as shown in Figure 5.3. The one-sided Jacobi is superior for very small dimensions such as 2×2 for $H > 21$. As the dimension starts to increase, the neural network requires significantly less number of operations for SVD computations. For instance, for $(m,n) = (20,16)$ and $(m,n) = (4,80)$ (these dimensions are often used for tensorial SVD implementations based on the one-sided Jacobi algorithm [73], [176]), computing the right singular vectors V using a shallow neural network requires less number of operations than using the one-sided Jacobi ($N < Nj$) for all values of $H \leq 70,000$ and $H \leq 800,000$ respectively. Such values of H

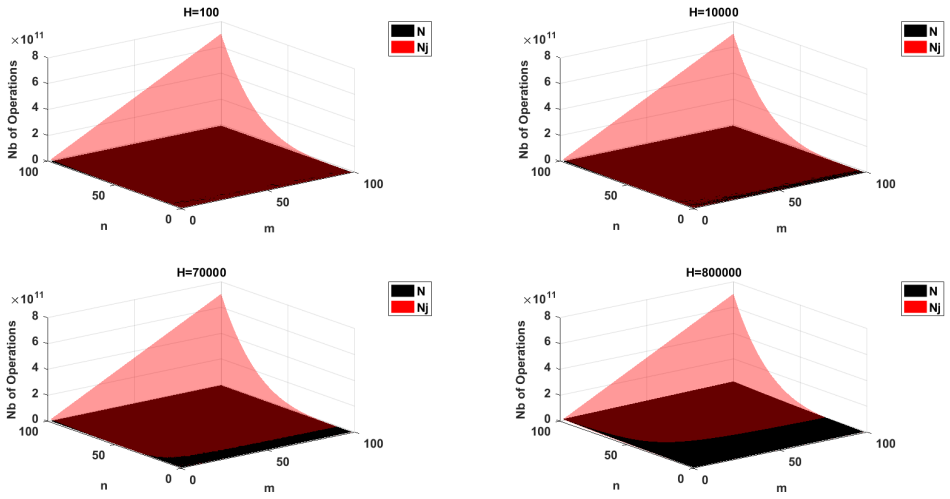


Fig. 5.3 Number of Operations required in one-sided Jacobi (Nj) and Shallow Neural Network (N), (m,n) are the matirx dimension and H is the hidden layer size.

are very large even for the largest existing neural networks. The number of operations (N) required for each activation function is presented in Table 5.1. The hidden layer activation function could be ReLU (Standard, LeakyReLU or Parametric), Sigmoid, or hyperbolic tangent (tanh), it is selected based on the trade-off between complexity and the required performance. In the output layer, only the hyperbolic tangent function can be used, due to the fact that the values of the singular vectors are bounded between -1 and 1.

Table 5.1 Complexity Assessment under different activation functions

Activation Function		**Number of Operations (N)**
Hidden Layer	**Output Layer**	
ReLU	Tanh	$(2H \times n)(m+n)+H+35n^2$
Leaky/PReLU		$(2H \times n)(m+n)+2H+35n^2$
Sigmoid		$(2H \times n)(m+n)+18H+35n^2$
Tanh		$(2H \times n)(m+n)+35H+35n^2$

5.4 SVD using Shallow Neural Networks

5.4.1 Network Structure

A regression model is targeted since the NN is needed to compute the singular vectors. For that, there are two possible categories to work on: (1) Classification NN that should be

modified to perform regression and re-trained [177] and (2) Regression NNs [178]. Although the classification accuracy achieved by the one-sided Jacobi TSVM could be obtained by an existing NN model from the two above mentioned categories, the main concern remains in the computational complexity of such model. Concerning the first category, one could choose Convolutional NN (CNNs), Multi-Layer Perceptron (MLP), Long-short Term Memory (LSTM), etc. Even the smallest models such as MobileNet [179], Shuffle Net [180], and EffNet [181] contains at least four layers. On the other hand, for the regression NNs, with only one hidden layer, a shallow network is considered to be the smallest possible regression NN model.

Figure 5.4 shows the proposed shallow neural network that is capable of computing the right singular vectors V. The network is composed of three fully connected layers: an input layer of size $m \times n$, a hidden layer of size H, and an output layer of size $O = n \times n$. f_h and f_O are the hidden and output activation functions respectively. Based on Table 5.1, the tangent hyperbolic function is used for f_O, while several functions can be used for f_h based on the training performance.

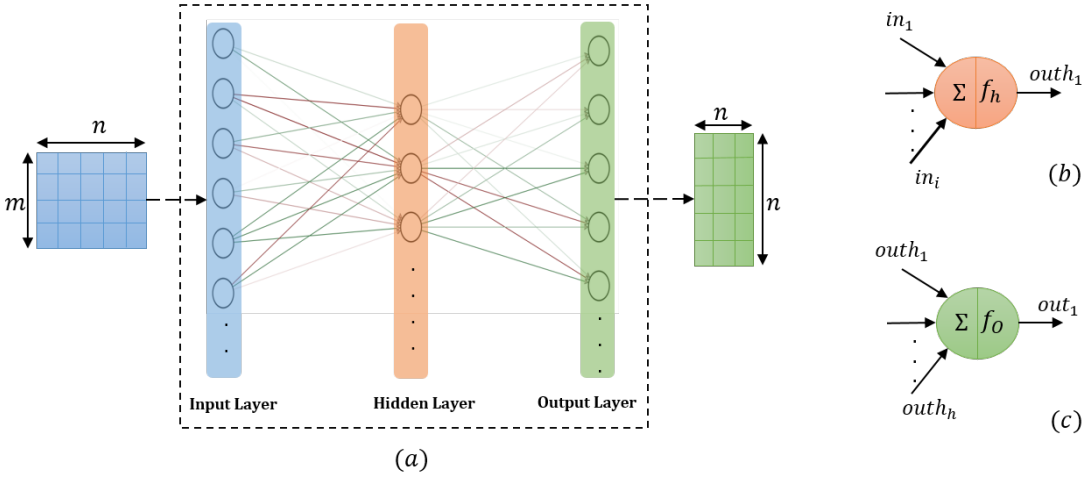


Fig. 5.4 Proposed Shallow Neural Network: (a) Overall Structure, (b) Hidden Layer Neuron, (c) Output Layer Neuron

5.4.2 Network Training

The touch modalities dataset from [5] is used for training. However, some modifications have been applied based on the following:

- Some participants recordings are noisy (see Figure 5.5(a)), thus their corresponding data has been removed from the training dataset.

- Since no particular indications were given to the participants in [5] about the pressure level, silent intervals (i.e. voltage readings from sensor taxels equals to zero, see Figure 5.5(b)) are observed in the recordings. These silent intervals will not help the neural network to learn new patterns and thus are removed. Specifically, all reading outside the timing interval [3.5, 7] are omitted.

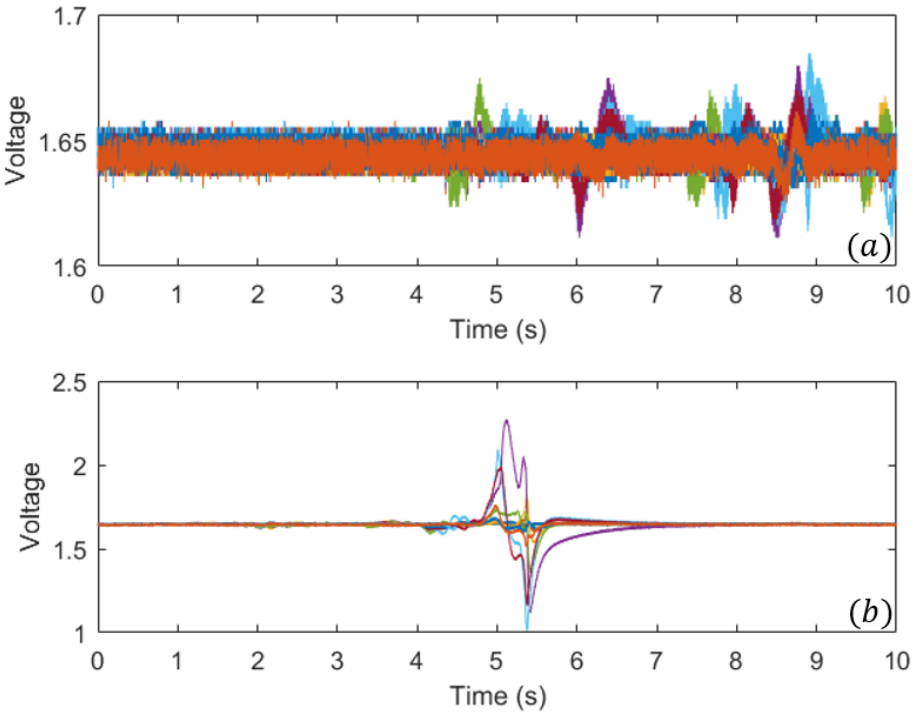


Fig. 5.5 Touch Modality with: (a) Noisy Readings, (b) Silent Intervals

Algorithm 3 summarizes the pre-processing technique applied to the dataset. The algorithm truncates each modality from $10s$ to $3.5s$ resulting in a tensor $T'(4 \times 4 \times 10,500)$.

Afterwards, subsampling is applied to obtain 20 readings ($P = 20$) from the 10,500 resulting in a final tensor $\phi(4 \times 4 \times 20)$. The obtained tactile tensor $\phi(4 \times 4 \times 20)$ is unfolded into three matrices $M(4 \times 80)$, $N(4 \times 80)$, and $P(20 \times 16)$. According to (3.3) each matrix could be decomposed into:

$$M_{4\times 80} = U_{4\times 80} \times \Sigma_{80\times 80} \times V^T_{80\times 80} \tag{5.10}$$

$$N_{4\times 80} = U_{4\times 80} \times \Sigma_{80\times 80} \times V^T_{80\times 80} \tag{5.11}$$

$$P_{20\times 16} = U_{20\times 16} \times \Sigma_{16\times 16} \times V^T_{16\times 16} \tag{5.12}$$

Algorithm 3: Pre-Processing Algorithm

Input: Tensor T of size (:,:,S),
Time Interval [a, b]
Sampling parameter P
Output: Sampled Tensor ϕ of size (:,:,P)
Let $v1 \leftarrow a \times S/10$
Let $v2 \leftarrow b \times S/10$
Let $S' \leftarrow v2 - v1$
Let T' be a Tensor of size (:,:,S')
Let $j = 0$
for $i \leftarrow$*v1* ***to*** *v2* **do**
  $T'(:,:,j) \leftarrow T(:,:,i)$
  $j++$
Let $k = 0$
for $i \leftarrow$*0* ***to*** *P* **do**
  $\phi(:,:,i) \leftarrow (P/S') * \sum_{i=k}^{S'/P+k} T'(:,:,i)$
  $k+ = S'/P$

Authors in [5] and [56] reported that for tensor SVD, only a small number of the columns of V is required to obtain acceptable classification accuracy when embedded in SVM. Using the three touch modality problems reported in section 5.2.3, the V matrices that resulted in the highest classification accuracy are: $V^T_{80\times4}$, $V^T_{80\times4}$, and $V^T_{16\times2}$. A total of 2240 matrices of dimensions 20×16 and 4×80 have been derived from the modified dataset. Then, the corresponding V matrices are generated using MATLAB. These matrices are divided into 80% for training, 10% for validation, and 10% for testing. Afterwards, we trained four networks whose input and output dimensions respectively are:

- *NN1*: 20×16 and 16×16.
- *NN2*: 4×80 and 80×80.
- *NN3*: 20×16 and 16×2.
- *NN4*: 4×80 and 80×4.

NN1 and *NN2* are designed for the computation of a generic matrix SVD, while *NN3* and *NN4* are designed for the computation of tensor SVD embedded in TSVM targeing tactile data classification where only a selected number of columns t of V is required. Each network model is trained using floating-point representation during both forward and backward propagation. The network is trained to export the right singular vectors V with the least

possible error margin compared to exact computations obtained via MATLAB. Since the proposed network is a regression model that outputs singular vectors, the performance is determined based on two metrics: (1) Mean Squared Error (MSE) and (2) Cosine Similarity (CS). These metrics are defined as:

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(V_i - \hat{V}_i)^2 \tag{5.13}$$

$$CS = \frac{1}{n}\sum_{i=1}^{n}(\frac{V_i.\hat{V}_i}{||V_i|| \times ||\hat{V}_i||}) \tag{5.14}$$

where V is the matrix generated from the neural network and $\hat{V}$ is the one generated from applying the SVD using MATLAB software. Thus, the training aims at finding a network model that achieves the lowest MSE (i.e. the elements v_i of the V and $\hat{V}$ matrices have similar values) and highest CS (i.e. the vectors V_i of the V and $\hat{V}$ matrices have similar direction i.e CS tends to 1).

The proposed neural network model is hand crafted and can be customized. The training process is used to tune the network hyperparameters [182] i.e. parameters that determines the network structure and training behavior (e.g. size of hidden layer H, learning rate) and parameters (e.g. weights). During training, the weights and biases of the network are randomly initialized, then updated using one of the below optimizers. As for the hyperparameters, the following settings have been tested:

- H = [10, 20, ... 200]
- Activation Function: Sigmoid, Tanh, ReLU, LeakyReLU, and PReLU with β: learned parameter through Channel-wise or channel-shared modes [183]
- Learning rate = [0.1, 0.01, 0.001, ... 10^{-5}]
- Optimizer : [SGD, Adam, Adadelta, RMSprop]
- Batch size = [50, 100, 150]

The four neural networks have been coded in Python using Tensorflow and Keras libraries. Then, they are trained on an ASUS PC equipped with an NVIDIA GTX 1650 graphics card with 4GB VRAM.

5.4.3 Network Performance

Figure 5.6 shows the training and validation MSE of the best network model for *NN1* and *NN2*. A training and validation MSE in the order of 10^{-3} and 10^{-4} is achieved via *NN1*

and *NN2*, respectively. Such performance is obtained using a model with the characteristics presented in Table 5.2. The two networks *NN1* and *NN2* and the exact one-sided Jacobi algorithm (based on the architecture presented in [166]) are used to compute the V matrix of 200 matrices (100 for each input dimension). The results show that both neural networks:

- can compute the V matrix of an input with a testing mean squared error of 2×10^{-3} and 4.5×10^{-4} for 16×16 and 80×80 output dimensions, respectively. Such results are comparable to the ones provided by exact one-sided Jacobi algorithm as shown in Table 5.2.
- achieves a low Root Mean Squared Error (RMSE) of 3×10^{-2} and 1×10^{-2} for 16×16 and 80×80 output dimensions, respectively. This is evident in Figure 5.7, where the exact V and the predicted V values are relatively close for both input dimensions.

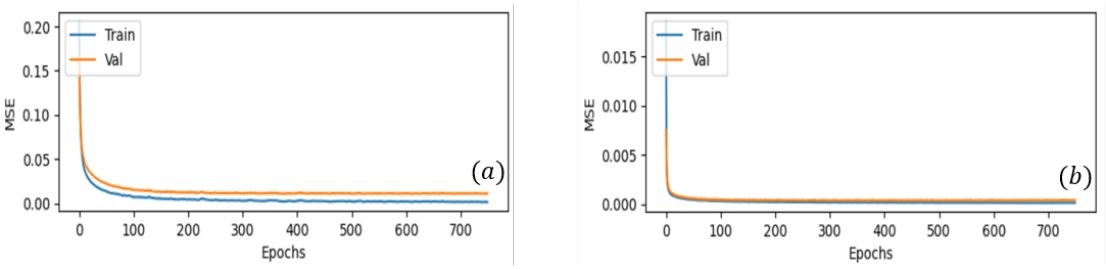


Fig. 5.6 Best Model Training and Validation MSE for (a) *NN1*, (b) *NN2*

Table 5.2 Neural Networks *NN1* and *NN2* Structure and Testing Performance

Neural Network	*NN1*	*NN2*
Input Layer Size	20×16	4×80
Hidden Layer Size	400	
Output Layer Size	16×16	80×80
Activation Function f_h	LeakyReLU with $\beta = 0.01$	
Learning Rate	0.001	
Batch Size	100	50
Epochs	300	200
Testing MSE	2×10^{-3}	4.5×10^{-4}
Testing MSE based on [166]	1.05×10^{-3}	2.9×10^{-4}

Figure 5.8 shows the best achievable MSE and CS for the networks *NN3* and *NN4* under the settings presented in Table 5.3. One noticeable observation is that the size of the hidden layer differs for the two input dimensions. This is due to the fact that *NN4* has to output 320

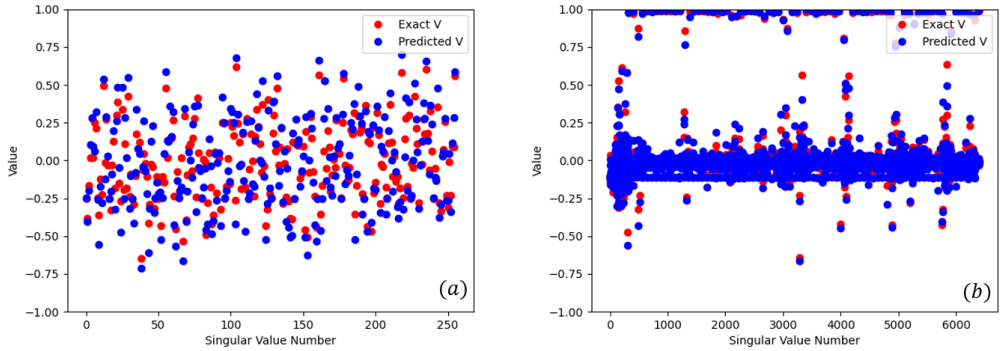


Fig. 5.7 V matrix Generation for Networks: (a) *NN1*, (b) *NN2*

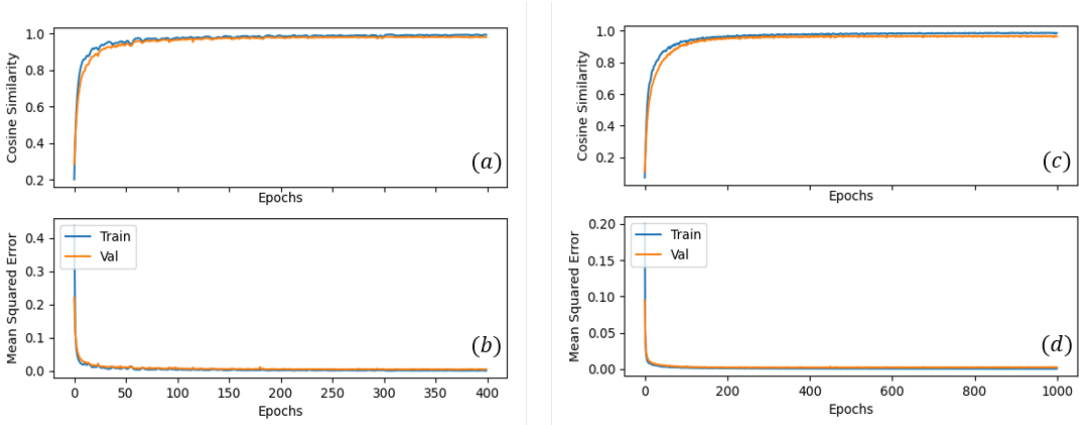


Fig. 5.8 Best Model Performance: (a) CS for *NN3*, (b) MSE for *NN3*, (c) CS for *NN4*, (d) MSE for *NN4*)

elements (80×4) for the input dimension 4×80 compared to 32 elements (16×2) generated by *NN3* for the input dimension 20×16, which justifies the longer training time required (higher number of epochs). However, the training can be shortened into 250 and 100 epochs for output dimensions 80×4 and 16×2 respectively. The obtained performance of *NN3* and *NN4* is compared to that of computing the SVD using the one-sided Jacobi algorithm based on the architecture presented in [166]. According to the comparison shown in Table 5.4, the proposed neural networks are capable of computing the right singular vectors V while: (1) providing low MSE and high CS during training, validation, and testing, and (2) achieving comparable performance in terms of MSE and CS to the exact computation using the one-sided Jacobi. This is evident for both output dimensions 16×2 and 80×4.

Since *NN3* and *NN4* are designed to be embedded in the TSVM architecture, the networks' complexity is a concern. Figure 5.9 shows the MSE and CS while testing *NN3* under different activation functions. Although using the hyperbolic tangent function leads to a model with

Table 5.3 Neural Networks *NN3* and *NN4* Structure

Neural Network	*NN3*	*NN4*
Input Layer Size	20×16	4×80
Hidden Layer Size H	40	140
Output Layer Size	16×2	80×4
Activation Function f_h	PReLU with Channel-shared β	
Learning Rate	0.001	
Batch Size	100	50
Epochs	400	1000

Table 5.4 *NN3* and *NN4* Performance Compared to one-sided Jacobi

Neural Network	*NN3*	*NN4*
Training MSE	9.45×10^{-4}	3.7×10^{-4}
Training CS	0.998	0.989
Validation MSE	9.63×10^{-4}	4.02×10^{-4}
Validation CS	0.982	0.969
Testing MSE	9.7×10^{-4}	4.18×10^{-4}
Testing MSE based on [166]	9.21×10^{-4}	3.88×10^{-4}
Testing CS	0.971	0.957
Testing CS based on [166]	≈ 1	≈ 1

the lowest MSE and highest CS, it imposes the highest computational complexity as reported in Table 5.1.

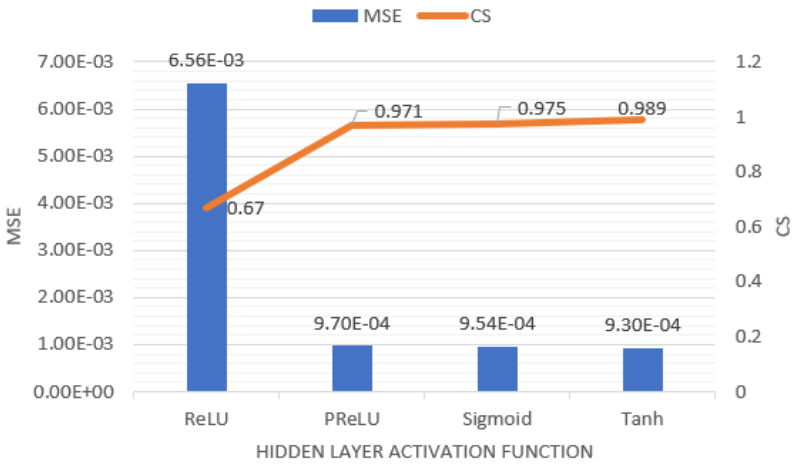


Fig. 5.9 Network Performance Under Different Activation Functions

Targeting the hardware implementation of NN-based TSVM, PReLU activation function has been adopted for the hidden layer as a trade-off between complexity and MSE/CS.

5.4.4 Hardware Implementation

This section presents the architecture adopted for the hardware implementation of the four shallow neural networks. The hardware architecture has been coded in C++, synthesized and implemented using Vivado/Vivado HLS 2020.1 targeting Virtex-7 FPGA device operating at 100 MHz. The input, weights, and biases are represented in 32-bit floating-point. To test and validate the hardware implementation, a C++/RTL co-simulation is performed in Vivado HLS to compare the results between the C++ simulation and the RTL implementation. Afterwards, the RTL implementation has been exported as an Intellectual Property (IP) to Vivado where the hardware resources and number of clock cycles are recorded. The time latency is computed as:

$$T = cc \times 1/f_{max} \tag{5.15}$$

where cc is the number of clock cycles in post-implementation timing simulation and f_{max} is the maximum operating frequency. As for power consumption, a post implementation functional and timing behavioral simulation is performed to generate a Switching Activity Interchange File (SAIF). This file is used to obtain a vector-based power estimation post-routing.

Figure 5.10 shows the proposed architecture for the implementation of the shallow neural networks ($t = n$ for *NN1* and *NN2*). For an input X of size L (one of the unfolded matrices), it outputs the V matrix using sequential operations. The outputs Y_h and Y_O corresponds to the equations (5.1) and (5.2), where f_h and f_O are the PReLU and the hard tangent hyperbolic activation functions respectively as shown in Figure 5.11. The input X and the weights are stored on-chip using BRAMs and the multiplier is fed from the BRAM to perform element-by-element multiplication of the input and weight values. Similarly, the multiplication result is fed to the adder and the bias values are read from on-chip BRAMs.

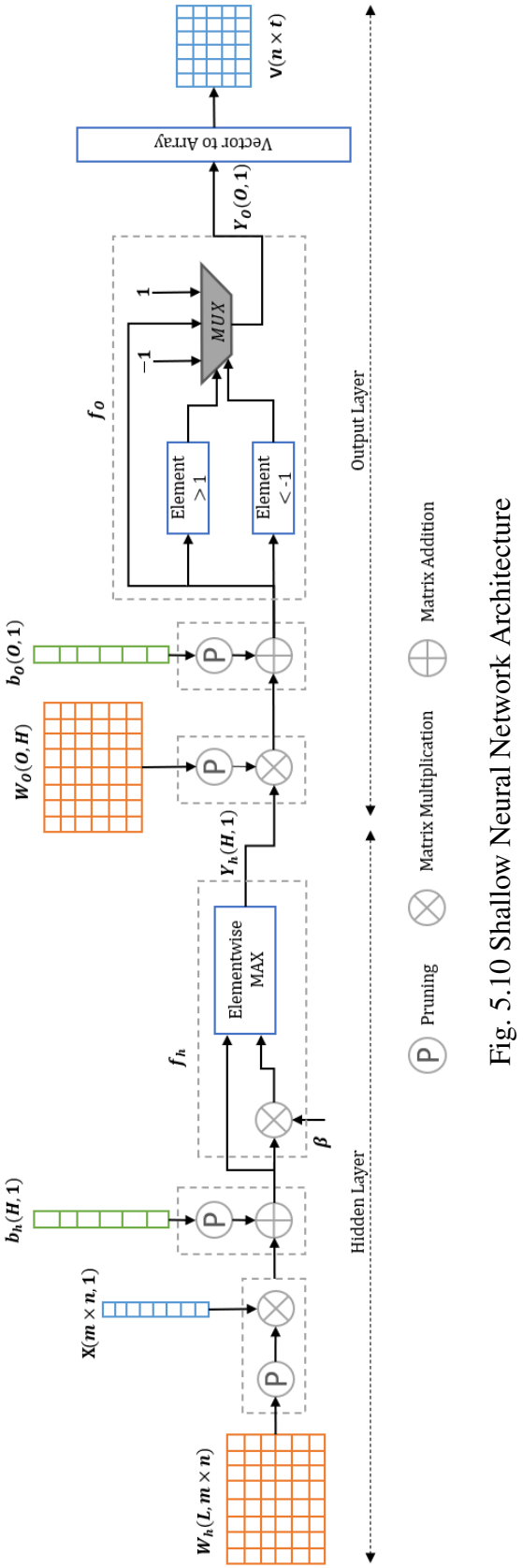


Fig. 5.10 Shallow Neural Network Architecture

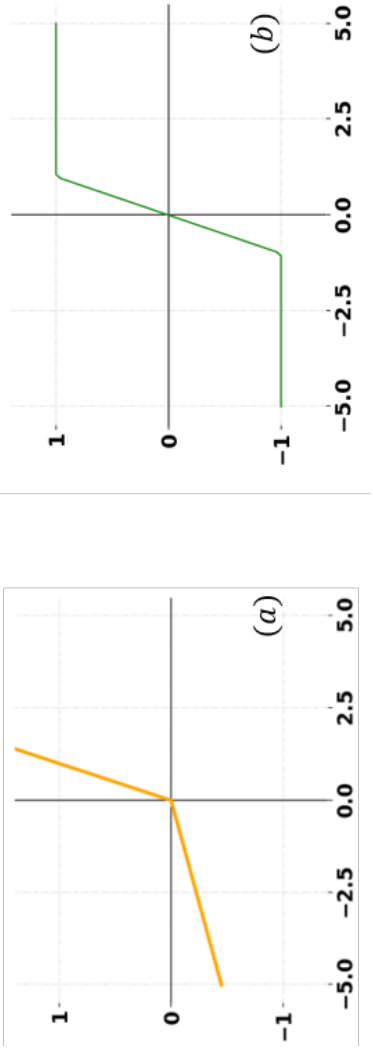


Fig. 5.11 Activation Functions: (a) Parametric ReLU, (b) hard Tangent Hyperbolic

The right singular vectors matrix V is obtained by transforming the output vector Y_O into a 2D array as shown in Figure 5.12 . The advantages of such architecture is that it: (1) imposes reduced hardware complexity with the use of hard-*tanh* instead of *tanh* without an accuracy loss, and (2) allows the use of network pruning without any loss in performance (MSE/CS). The weight and bias matrices obtained from the offline training phase have been analyzed to identify neurons with very low weight/bias values. These neurons could be removed without affecting the network performance during inference. Thus, pruning is applied on matrix multiplication/addition by skipping operations where $W[i], b[i] \leq 10^{-4}$.

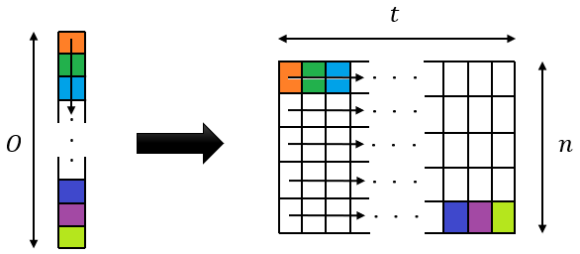


Fig. 5.12 Vector Y_O to Array V Transformation

Table 5.5 presents the implementation results of *NN1*, *NN2*, and state-of-the-art one-sided Jacobi architecture. The results show that both *NN1* and *NN2* require a lower average hardware resources up to 86%.

Table 5.5 *NN1* and *NN2* Implementation Details on Virtex-7 FPGA

	NN1	**Jacobi-SVD [166]**	***NN2***	**Jacobi-SVD [166]**
Input Dimension	20×16		4×80	
BRAM	1	4	16	5
DSP	5	30	5	30
LUT	761	13086	816	13357
FF	716	8788	773	9205
f_{max} **(MHz)**	118.13	109.5	118.13	109.5
Time Latency	8 ms	40 ms	0.25 s	4.7 s
Speedup	$5\times$		$18.8\times$	
Average Resources Reduction(%)	86%		12.2%	

To output the V matrix, speedups between $5\times$ and $18.8\times$ have been recorded. Furthermore, two observations could be highlighted: (1) both architectures use the same number of DSPs regardless of the matrix dimension, which is compensated with the use of LUTs; with a $6\times$ reduction in favor of the *NN1*/*NN2*. (2) The network *NN2* demands an increased usage of BRAMs for the input matrix 4×80. This is due to the fact that the weight and bias

matrices are stored on chip. In case of an input matrix of size 20×16, this is not an issue since the dimensions of the weight, bias, and output V matrices are lower than that in the case of 4×80 matrix (e.g. for 4×80 matrix, the W_2 matrix is of size 6400×400 compared to 256×400 for 20×16 input matrix).

Table 5.6 shows the implementation details for the SVD computation of a $4 \times 4 \times 20$ input tensor using shallow neural networks (i.e. NN3 is utilized once to compute the SVD of the matrix P, while NN4 is utilized twice to compute the SVD of the matrices M, N) compared to the one-sided Jacobi based on the architecture presented in [73].

Table 5.6 Implementation Results for Tensor SVD Computations

Architecture	Neural Network	one-sided Jacobi
BRAM	102	88
DSP	32	105
FF	3714	29277
LUT	4905	43258
Time Latency	14.5 ms	4.7 s
Power Consumption	0.45W	1.35W

The obtained results show that using neural networks for SVD computations allows for a 324× speedup with an average resources and power reductions of 58% and 67% respectively. Another observation is that the neural network architecture uses slightly more BRAMs. This is due to the fact that the weights and biases matrices obtained from network training are mapped into BRAMs and are not saved on an external memory. Knowing that the Virtex-7 FPGA is used for implementation to have a credible comparison with the state-of-the-art, the obtained results show that the proposed neural network for SVD computations is adequate to fit in a resource-limited platform such as the Zynqberry. This is not possible for the implementation of the one-sided Jacobi targeting large matrix dimensions.

5.5 TSVM based on Shallow Neural Networks

5.5.1 Proposed Architecture

The neural networks *NN3* and *NN4* have been embedded into the cascade architecture of the tensorial SVM presented in [73]. The new NN-based TSVM architecture is presented in Figure 5.13. The "NN Memory" contains the weights and biases matrices of the designed neural networks. The "SVM Memory" contains the singular vector training matrices. k_1, k_2, and k_3 are the three kernel factors obtained using (3.4).

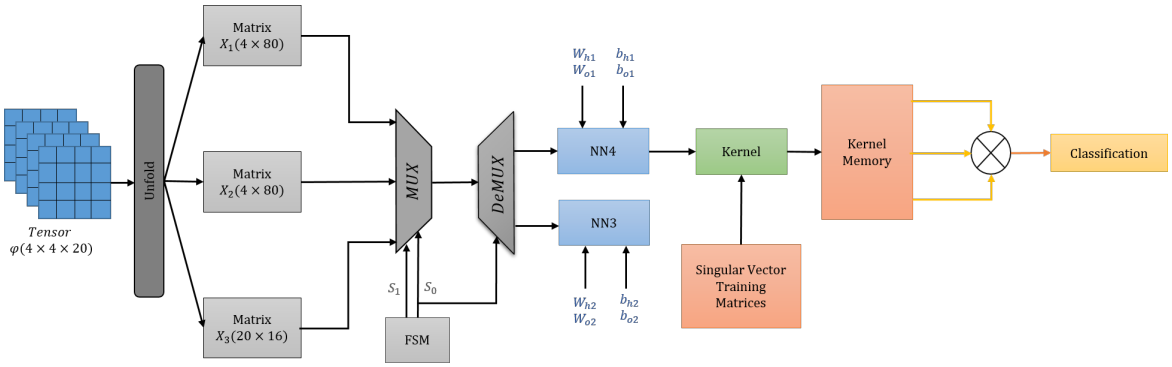


Fig. 5.13 Neural Network based TSVM Cascade Architecture

The architecture performs the SVD computation of the three unfolded matrices using the proposed *NN3* and *NN4* neural networks. Table 5.7 shows the different operating modes in the cascade architecture. For $S_0S_1 = 00$, the first unfolded matrix X_1 is selected and *NN4* is activated, then for $S_0S_1 = 01$, the second unfolded matrix X_2 is selected and *NN4* is utilized. As for $S_0S_1 = 10$, the third unfolded matrix X_3 is selected and *NN3* is activated. When active, each network computes the right singular vector matrix V of each of the unfolded input matrices. The obtained V matrices along with the ones exported from the training phase are used to compute the kernel factors as depicted in (3.5), which are required to output a classification decision as shown in (3.6).

Table 5.7 NN-based TSVM Operating Modes

Control		**MUX**	**Active**
S_0	S_1	**Output**	**Network**
0	0	X_1	*NN4*
0	1	X_2	*NN4*
1	0	X_3	*NN3*
1	1	-	-

5.5.2 Implementation Results

Table 5.8 presents the implementation details of both the NN-based TSVM and Jacobi-based SVM for $N_t = 200$ and $N_c = 2$. The energy per classification is computed as $E = P \times T$ where P is the dynamic power consumption and T is the time latency.

Results show that replacing the one-sided Jacobi algorithm with a shallow neural network in the architecture of the TSVM leads to faster classification time up to 131×. The NN-based TSVM and Jacobi-based TSVM recorded 0.9 W and 1.8 W respectively, thus a 50% reduced

Table 5.8 Implementation Results for NN-based TSVM on Virtex-7

Architecture	NN-based TSVM	Jacobi-TSVM
BRAM	105 (7.14%)	91 (6.19%)
DSP	133 (3.7%)	206 (5.72%)
FF	11975 (1.38%)	39047 (4.51%)
LUT	20427 (4.7%)	60100 (13.87%)
Time Latency (ms)	36	4730
Energy per classification (mJ)	6.28	600

power consumption is achieved. This leads to 88% reductions in the energy per classification factor. The NN-based TSVM also requires 39% less average hardware resources. Following these results, three main observations could be noted: the proposed NN-based TSVM: (1) is capable of real-time classification within 36 ms ($time \leq 400ms$ [4]), (2) achieves real-time classification using a cascaded architecture, which was not possible using the Jacobi-based TSVM as reported in [73]. The latter has been the main reason for using the parallel architecture which has lead to high power consumption, and (3) offers the reductions in resources and energy per classification at the expense of increased memory requirements to store the weights and biases matrices compared to Jacobi-based TSVM.

5.5.3 Performance Verification

The NN-based TSVM implementation is verified using the four classification problems mentioned in section 5.2.3. Table 5.9 presents the classification accuracy achieved by the proposed NN-based TSVM in comparison with existing methods targeting the same touch modality classification. The classification accuracy of different methods is tested using a dataset with 30 unseen samples.

Table 5.9 Touch Modality Classification Using NN-based TSVM in Comparison with existing methods

Problem	**Classification Accuracy (%)**							
	NN-based TSVM	**Jacobi TSVM [5]**	**RLS [5]**	**k-NN (k=3) [184]**	**DCNN [32]**	**LSTM [185]**	**GRU [185]**	**TSVM-IRCK [55]**
A	90	90	90	68	NA	NA	NA	NA
B	83.3	86.3	89.5	64				
C	80	83.3	75.5	89.6				
D	71	71	73.3	NA	76.9	74.51	73.4	77

Using neural networks to compute the right singular vectors V provides approximate values compared to the exact one-sided Jacobi. However, this resulted in acceptable classification accuracy with only 3% loss in the worst case. This is evident in the comparable MSE/CS of both architectures as presented in Table 5.4. Compared to other methods, for binary classification (A, B, and C), the proposed NN-based TSVM shows a worst case of 6%

loss compared to RLS for Problem B and a 5% better accuracy for Problem C, and 9.6% loss compared to kNN for Problem C while providing up to 20% accuracy increase in Problems A and B. Problem C has been identified as very challenging for TSVM in [5], it has been solved in [184] using k-Nearest Neighbor (kNN). For multi-class classification (Problem D), NN-based TSVM achieves an accuracy comparable to Jacobi-TSVM and RLS, with a 7% worst case loss compared to Deep Convolutional Neural Network (DCNN) and TSVM with Ideal Regularized Composite kernel (TSVM-IRCK).

5.6 Scalability Assessment

In order to quantify the scalability of the NN-based TSVM hardware complexity (resources and time latency), two cases are assessed: (1) scalability of the shallow neural network, and (2) scalability of the NN-based TSVM. The former is studied by varying the hidden/output layer size and tuning the network to achieve the same MSE/CS reported in Table 5.4. The latter is performed by increasing the number of training tensors while maintaining the overall classification accuracy of the NN-based TSVM as reported in Table 5.9.

5.6.1 Case 1: Scalability of the Shallow Neural Network

The scalability of the neural network depends on the size of each layer and the activation function in use. Through Figure 5.3, an insight about the number of operations with respect to the dimensions (i.e. m, n, and H) could be learned for a certain application. To assess the scalability of the proposed NN architecture, the hidden and output layer sizes are varied. L is chosen so that the network maintains the same MSE/CS reported in Table 5.4. O is derived from the dimension of the V matrix of the unfolded matrices obtained from the input tensor $4 \times 4 \times 20$. Figure 5.14 presents the hardware resources and the time latency with respect to hidden and output layer sizes for a three-layer (input, hidden, output) shallow neural network for the SVD computation of an input matrix. The obtained results are recorded when the network achieved a comparable MSE/CS to those reported in Table 5.4. Analyzing the graphs leads to several observations:

- The number of required FFs and LUTs is not uniform (see Figure 5.14(a),(b)). For instance, a similar number of FFs/LUTs is required for networks with 140 and 400 neurons in the hidden layer with the same output layer size. This could be justified with the pruned cascaded architecture where resources are shared for blocks with similar functionality.

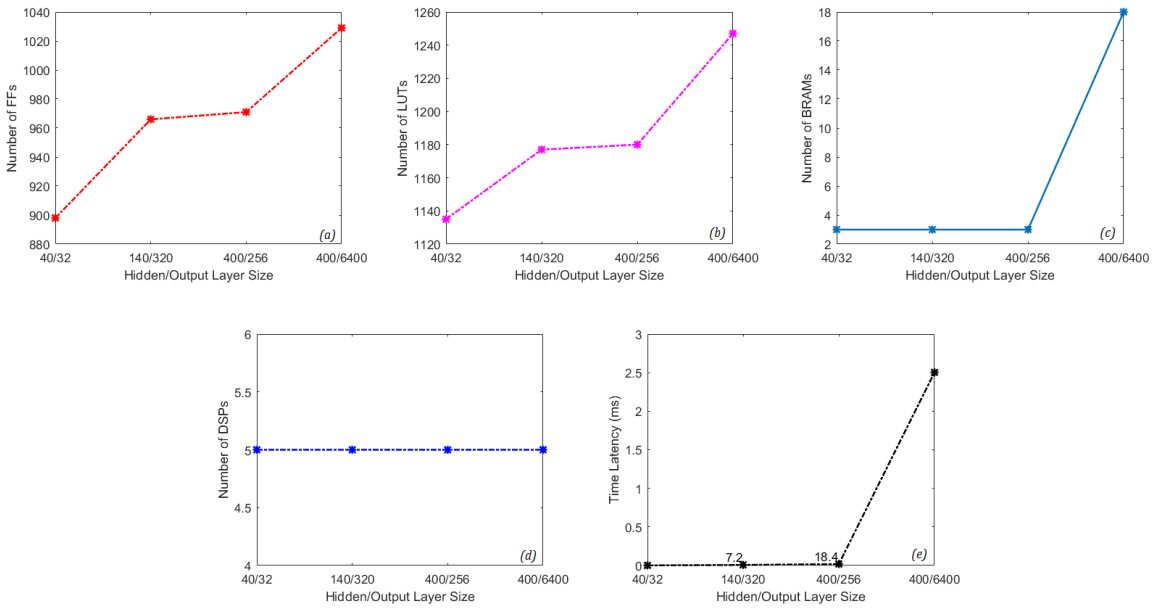


Fig. 5.14 Scalability of Shallow Neural Network for varying the hidden/output layers size

- Memory requirements in terms of BRAMs starts to increase once reached an output layer size of 80×80 with 400 neurons in the hidden layer (see Figure 5.14(c)). This is justified since the size of the weight and the bias matrices increase in such cases, which requires more memory storage.

- As shown in Figure 5.14(d), regardless of the input/hidden/output layer size of the network, the number of DSPs is constant for the proposed architecture.

- The SVD computation time is relatively short until reaching a large output layer size as shown in Figure 5.14(e). This is due to the longer operations required to perform matrix multiplication/addition. However, according to the comparison in Section 5.3.2, this is faster than using the one-sided Jacobi as long as $H \leq 70,000$ ($H \leq 800,000$) for 20×16 (4×80) matrices.

The presented scalability assessment supports the use of these networks for SVD computations as an efficient solution especially for large matrix dimensions. Hence, the proposed idea could be extended into other applications via a two-stage approach as shown in Figure 5.15:

- *Stage 1:* Unfold all the tensors ϕ_i in a dataset into 3 matrices. Then, find the V matrix for each of the unfolded matrices using MATLAB or other software. For the majority of the applications, a tensor has the same first two dimensions (e.g. image, touch

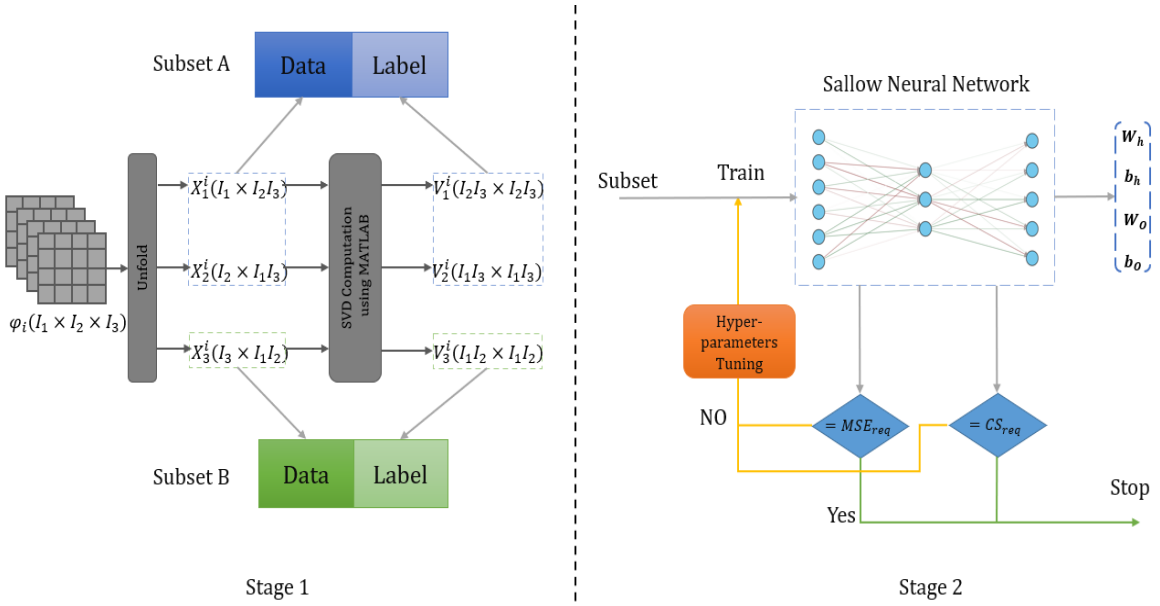


Fig. 5.15 SVD Computation Approach via a Shallow Neural Network

modality) hence, two of the generated matrices will have the same dimension hence can be grouped in a subset A. The remaining matrix and its corresponding V matrix will be added to a subset B.

- *Stage 2:* For each of the subsets, a shallow neural network is to be designed. Start with random hyperparameters for the initial model, then tune it using the generated subset to reach the required MSE and CS. Once, the best model is found, the weights and biases matrices could be exported and used by the architecture in Figure 5.10. For complexity tuning, one could modify the pruning rule while preserving the required performance metric imposed by the application.

5.6.2 Case 2: Scalability of NN-based TSVM

To study the scalability of the proposed NN-based TSVM, the number of training tensors has been varied between 200 and 900 and the implementation requirements are recorded once the NN-based TSVM achieves a comparable accuracy to the one presented in Table 5.9. According to the results obtained in Figure 5.16:

- The required hardware resources (FFs, LUTs, BRAMs) are slightly increased with the increase of the number of training tensors. In case of BRAMs, a steeper slope is observed which is due to the adoption of NN that requires the storage of weights and biases matrices.

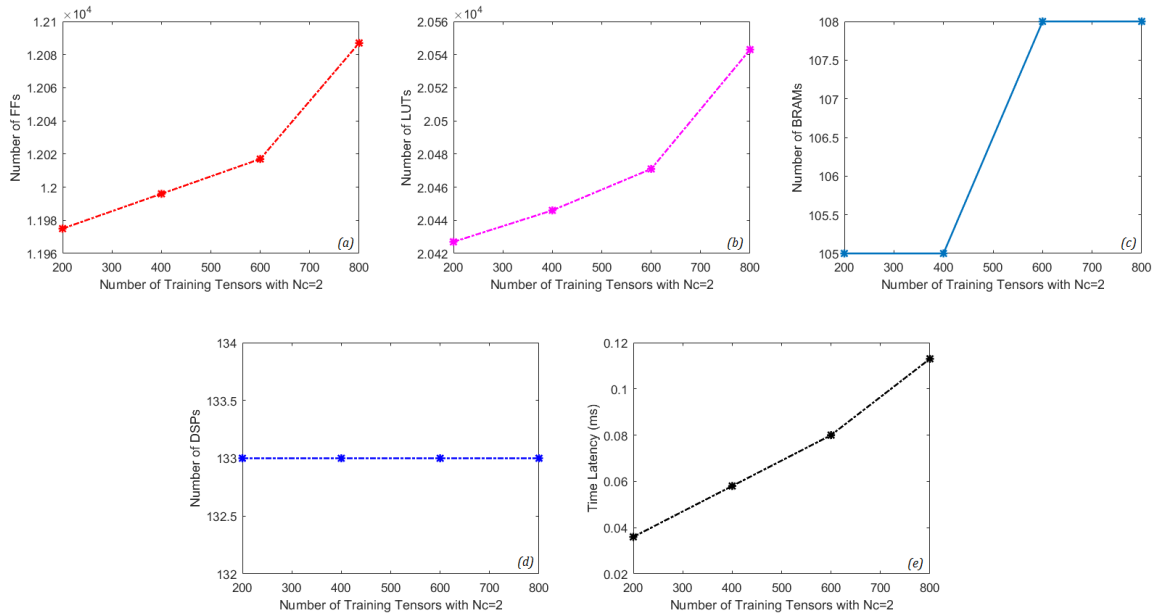


Fig. 5.16 Scalability of NN-based TSVM for binary classification ($Nc = 2$) and variable number of training tensors

- The number of required DSPs is contsant for each size of training tensors.
- The proposed implementation is capable of real-time classification even after 4.5× increase in the number of training tensors.

Compared to the scalability study of the Jacobi-based TSVM presented in [75]*, Figure 5.17 shows that the proposed approach complexity versus the number of training tensors presents a reduced slope. For instance, the Jacobi-based TSVM requires 29% increase in the number of FFs when the number of training tensors is doubled. Using the NN-based TSVM, an increase of less than 1% in FFs is noticed. This is mainly due to two reasons: (1) the neural network requires significantly less resources than that of the one-sided Jacobi, and (2) the NN-based TSVM is a cascaded implementation i.e. blocks are being re-used for implementation while increasing the time latency. In [75], the architecture is based on parallel computation due to their time constraint of real-time classification. The latter is assured using the proposed cascaded architecture for all of training tensors sizes.

The importance of the presented work lies in the ability to scale such architecture for processing larger number of samples while respecting the constraints of the application. When scaled up, the designed NN-based TSVM could enable intelligence on smaller platforms (e.g. Zynqberry) if two issues are tackled. The first issue is reducing the number of DSPs: this could be achieved by using some approximate computing techniques [95] or using LUTs-only

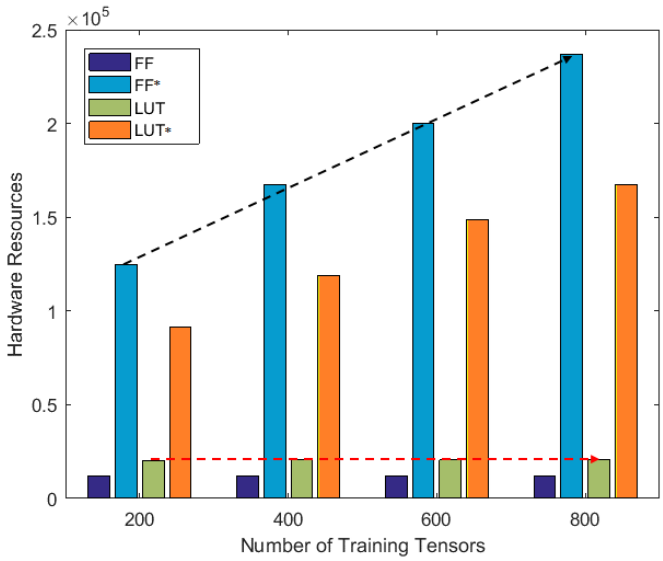


Fig. 5.17 Scalability Comparison with Existing Methods

custom core for matrix operations. The second issue is reducing the number of BRAMs: this could be achieved by further pruning of the weight/bias matrices as long as the application performance is not highly affected. Another method is to offload these matrices completely to external DRAM. This imposes additional timing overhead. However, authors in [44] have presented a strategy to overcome such design challenge.

5.7 Conclusion

A shallow neural network architecture for the SVD computation of tensorial inputs is presented. The architecture achieves comparable performance to the state-of-art solutions while imposing significant reductions in the implementation requirements. Once embedded in the SVM architecture, the NN-based TSVM is capable of delivering faster touch modality classification time up to 131$\times$ using a cascade architecture. The latter is characterized by a 39% and 88% decrease in the resources and energy per classification respectively compared to the architecture presented in [73] targeting the same application. Moreover, the proposed NN-based SVM obeys the constraints imposed by the tactile data processing application e.g. small size, real-time response, and low power consumption. The encouraging scalability results present the first effective trial for designing an efficient embedded processing unit for an Electronic Skin (e-skin). A unit that is capable of delivering real-time performance with relatively acceptable power consumption without the need for high performance platform or multi-core devices.

Chapter 6

A Hybrid Precision Architecture for an Efficient Binary Convolutional Neural Network Accelerator

6.1 Introduction

Convolutional Neural Networks (CNN) are a promising solution in many application domains such as Internet of Things (IoT), image processing, tactile processing, etc. However, the computational complexity and memory requirements are the main challenge in the deployment of CNNs on resource-limited devices for energy-constrained applications [186]. For instance, the VGG-16 network contains about 140 million 32-bit floating-point parameters and implements 1.6×10^{10} arithmetic operations [76]. There have been numerous efforts on the complexity reduction of CNNs such as network pruning [187], knowledge distillation [188], and weight quantization [189].

Quantization of CNNs may cause an information loss especially if it is applied to the extreme using 1-bit representation i.e. binarization. To address this issue, a variety of methods have been proposed in recent years [76]. These methods aim to: 1) minimize the quantization error, for instance by only quantizing the weights, 2) improve the network loss function to adapt to the binary values propagating through the network, and 3) reduce the gradient error by the adjustment of the Back Propagation (BP) training algorithm to adapt with binarization functions. Among these methods, minimizing the quantization error is the most used technique since it leads to relevant memory saving and complexity reductions [76], [189], [77].

In this chapter, a new architecture based on hybrid precision representation is proposed as a trade-off between the reliability of CNNs and the low complexity of Binary Convolution Neural Networks (BCNN). The architecture adopts binarization of hidden layers and 32-bit floating-point for the first and last layer with binary weights. A design methodology is provided on how to select the network topology, placement of binarization layers, and training process. The network is designed and trained using Larq framework [88], which is an extension of Tensor-Flow that offers a library to design, train, and deploy quantized/binarized CNNs. The proposed Hybrid-precision Binary Weight Network (H-BWN) achieved more than 35% accuracy increase in classifying touch modalities compared to traditional BCNN topology. The H-BWN requires less than 5 KB of storage requirements achieving an efficient architecture that fits in a wide range of microcontrollers (e.g. STM32F0x2). When implemented on Zynq-7010 platform, the H-BWN accelerator provided a real-time classification within 0.8 ms with a 42.4 μJ energy per classification. Compared to exiting solutions, H-BWN achieved higher classification accuracy with an energy reduction up to 99% accompanied with a speedup up to 6875×.

6.2 Binary Convolution Neural Networks Overview

The convolution operation in a convolution neural network can be expressed as:

$$Y = \sigma(\mathbf{w} \otimes \mathbf{a}) \tag{6.1}$$

where **w** and **a** represent the weight and activation tensors given to a network layer, respectively. σ is a non-linear activation function, Y is the output tensor and $\otimes$ represents the convolution operation. The latter is composed of matrix multiplication and addition that consists of a large number of floating-point operations. A BCNN uses 1-bit representation for the floating-point weights/activations. Hence, three configurations have been reported in the literature: (1) Binary Neural Network (BNN) where both weights and activations are binarized, (2) Binary Weight Network (BWN) where only weights are binarized [83], and (3) Binary Activation Network (BAN) where only activations are binarized [76].

–Forward Propagation:

The weights and/or activations are binarized using a binarization function defined as:

$$q(\mathbf{w}) = \alpha b_{\mathbf{w}} \tag{6.2}$$

$$q(\mathbf{a}) = \beta b_{\mathbf{a}} \tag{6.3}$$

where $b_{\mathbf{w}}$ and $b_{\mathbf{a}}$ are the binary weights and activations respectively, with their corresponding scalars α and β. A widely used binarization function is the *sign* function defined as:

$$sign(x) = \begin{cases} +1 & x \geq 0 \\ -1 & otherwise \end{cases} \tag{6.4}$$

Substituting 6.2 and 6.3 in 6.1, the convolution operation in the forward propagation of a BCNN can be written as:

$$Y = \sigma(q_w(\mathbf{w}) \otimes q_a(\mathbf{a})) = \sigma(\alpha\beta(b_{\mathbf{w}} \odot b_{\mathbf{a}})) \tag{6.5}$$

where $\odot$ is the inner vector product. Hence, the convolution operation in a BCNN can be performed efficiently using a bitwise XNOR-Bitcount operation as shown in Figure 6.1.

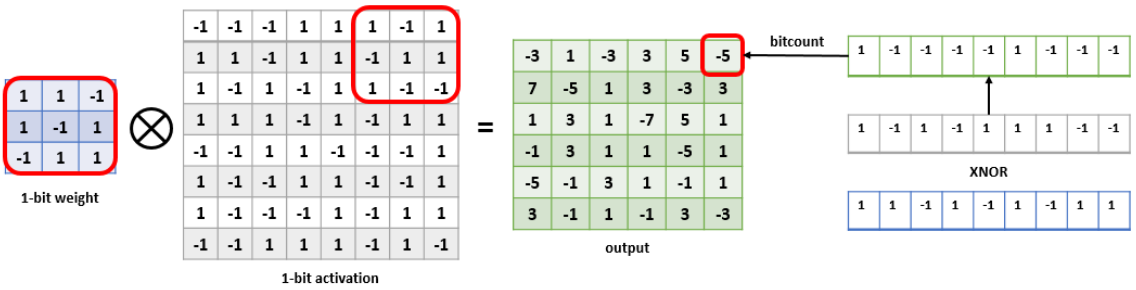


Fig. 6.1 Convolution in Binary Convolution Neural Networks

–Backward Propagation:

Training a BCNN using the commonly used backward propagation algorithm based on the gradient descent is not straight forward due to the fact that the binarization function is not differentiable (e.g. *sign* function). Hinton *et. al* proposed a technique called Straight Through Estimator (STE) to solve the gradient descent problem [190]. The STE function is defined as:

$$clip(x,-1,1) = max(-1,max(1,x)) \tag{6.6}$$

An approximation of the *clip* function is used in practice since the network can't be updated in back propagation if the absolute value of the activations is greater than one.

6.3 Hybrid-Precision BWN Model

6.3.1 Design Methodology

Based on the findings illustrated in Chapter 2, the aim is to design a BWN with a hybrid precision in each layer. The proposed H-BWN is targeting the touch modality classification problem based on the dataset collected in [5], hence a customized topology is adopted where existing ones are not specifically designed to provide acceptable accuracy with low computational complexity for the given task [32]. The H-BWN should be able to classify a tactile input as one of three modalities: "Rolling", "Sliding" and "Brushing". Before designing the network, a set of modifications have been applied to the original dataset:

- Each touch sample has been truncated from 10 to 3.5 seconds by eliminating all the readings outside the interval [3.5s, 7s] where no touch is present (See Figure 6.2 (a), (b)). Hence, the new touch sample size is reduced from $\phi(4\times4\times30,000)$ to $\phi(4\times4\times10500)$ as shown in Figure 6.2 (c).

- Subsampling is applied on the truncated touch sample to obtain a new tensor of size $\phi(4\times4\times8)$ (see Figure6.2 (d)) where each $P=1312$ readings are averaged into one reading according to the following equations:

$$\begin{cases} T(:,:,i) = (1/P) * \sum_{i=k}^{P+k} \phi(:,:,i) \\ k+ = P \end{cases} \tag{6.7}$$

  starting with $k=0$ and iterating $i=1,2...,8$.

- Due to the accumulated precision loss between the layers of a BCNN, the network is usually trained for a longer time (i.e larger number of epochs). To achieve this, a larger dataset is required. The tactile dataset contains 780 samples after applying the above modifications. Thus, data augmentation is applied on the dataset, specifically a rotation by 90 degree as shown in Figure 6.2 (e). This resulted in a dataset with 2*780 = 1560 samples.

For an input $\phi(4\times4\times8)$, the H-BWN design methodology is derived as follows:

- The number of the convolution and fully connected layers is determined as the lowest possible value to achieve a comparable classification accuracy to the existing similar solutions targeting the same touch modality problem [32], [5]. Only the size of the last fully connected layer is pre-defined to three neurons. Each neuron corresponds to one of the three modalities.

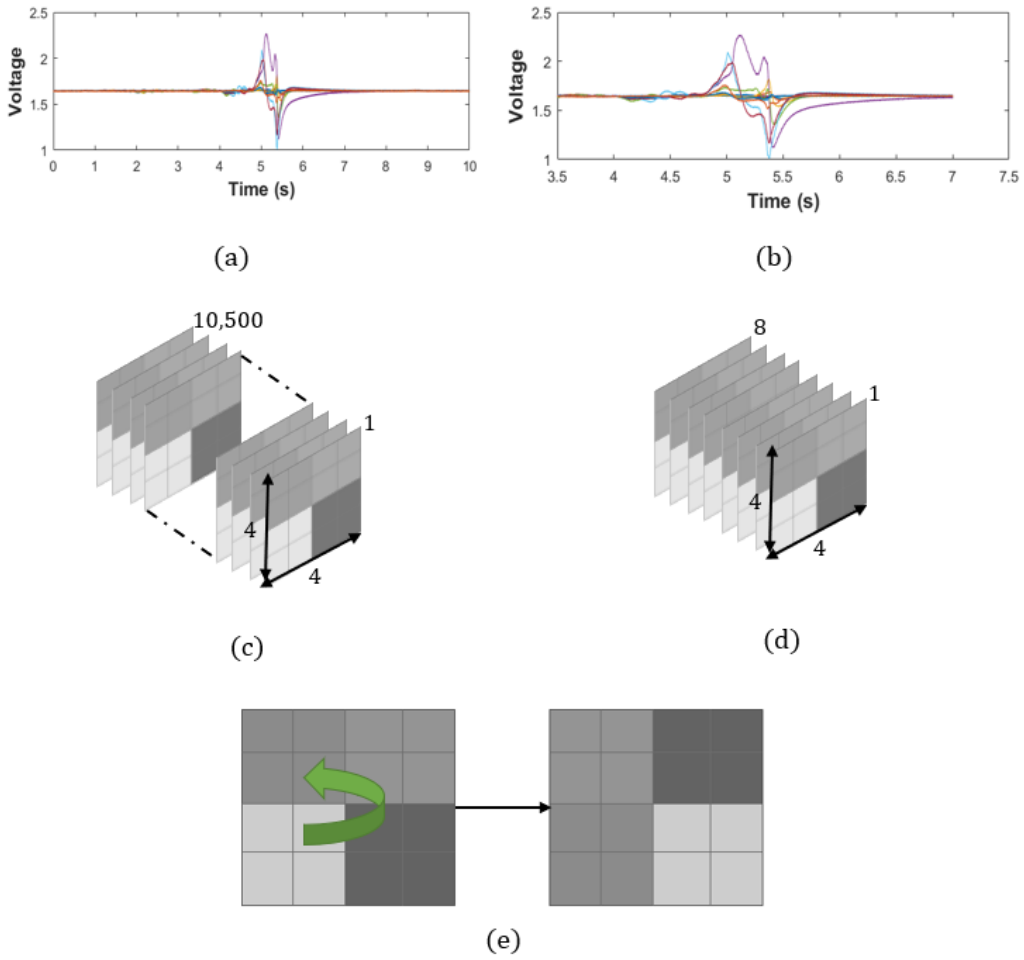


Fig. 6.2 Data Pre-processing: (a) Touch Modality, (b) Truncated Touch Modality, (c) Tensor Representation, (d) Sampled Tensor, (e) Data Augmentation

- Starting from an input size $4 \times 4 \times 8$, the maximum number of pooling layers that could be used is two layers, then the output of the layer will be a single element. Thus, to keep as much features as possible, a max pooling layer is used only once. The adopted pooling size is 2×2.

- Due to binarization, the output of each layer is unbalanced, thus a batch normalization layer is usually inserted to rectify the data [76]. After this normalization, the data obeys a stable distribution. Thus, to keep the mean and variance within a reasonable range, batch normalization is used in all layers resulting in a much smoother training process.

- Since the input size is small (4×4) compared for example an image (32×32), convolution is applied with padding in order to keep as much features as possible especially

the ones at the edge. Hence, the output of each convolution layer has the same size as the input of that layer.

- In a traditional convolutional neural network, the first layer should extract as much features as possible from the input before max pooling is applied, and the last layer should formulate a classification decision. Followed by the finding in section 6.2, these two layers have been designed with hybrid precision where activations are quantized and kernels are binarized. For the hidden layers, a complete binarization is performed. Hence, the base model used for training is presented in Figure 6.3.

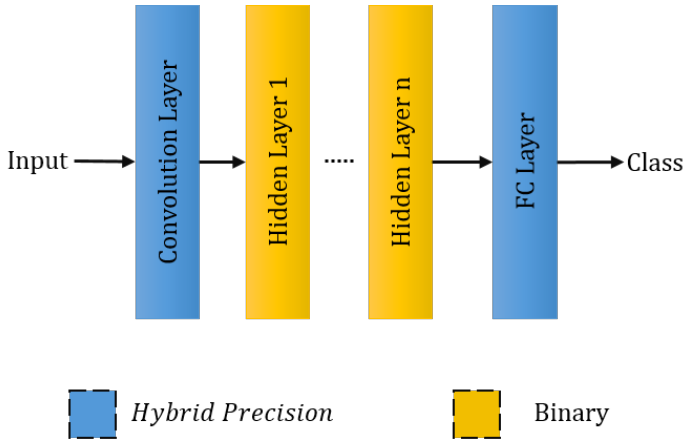


Fig. 6.3 Hybrid Precision Neural Network Model

6.3.2 Network Training

The modified tactile dataset is used to train six networks: BNN, BWN, BAN, and their hybrid precision counterparts (i.e. H-BWN, HBWN, and H-BAN) based on the model shown in Figure 6.3. The dataset has been divided into 5 folds where each time the network is trained using 4 folds and tested using the remaining one. The classification accuracy is determined as the average of the 5 folds. This process has been repeated 10 times to determine the best batch-size and number of epochs required to achieve the highest possible accuracy. The networks have been modeled in Python using the Larq framework [88].

Table 6.1 shows the quantizer type for the different network configurations. We investigated the use of SteSign, SwishSign, SteTern, DoReFa, and Approx-Sign (these quantizers are available online at https://docs.larq.dev/larq/api/quantizers/). The "Approx-Sign" quantizer shown in Figure 6.4 has been adopted as it resulted in the highest classification accuracy for the studied binarized neural networks. In forward propagation, the Approx-Sign is defined as:

Table 6.1 Quantizers for BNN Training

	BNN	BWN	BAN
Input Quantizer	Approx-Sign	None	Approx-Sign
Kernel Quantizer	Approx-Sign	Approx-Sign	None
Kernel Constraint	Weight clip	Weight clip	None

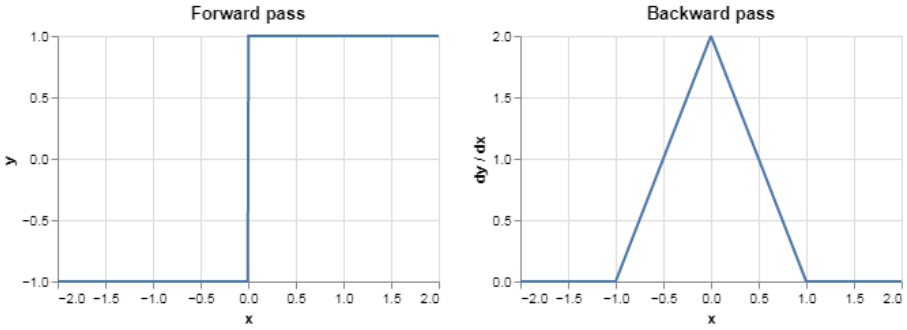


Fig. 6.4 Approx-Sign Quantizer in Forward and Backward Propagation

$$q(x) = \begin{cases} -1 & x < 0 \\ 1 & x \geq 0 \end{cases} \tag{6.8}$$

As for the gradient it is estimated using the following equation:

$$\frac{\partial q}{\partial x} = \begin{cases} 2 - 2|x| & |x| \leq 1 \\ 0 & |x| > 1 \end{cases} \tag{6.9}$$

The "weight_clip" kernel constraint represents the clip function defined in 6.6. The networks has been trained on an Intel i-7 based PC equipped with NVIDIA GTX 1650 graphics card. Adam [191] with a learning rate of 0.01 is used as an optimizer with “categorical-crossentropy” loss function.

6.3.3 Network Assessment

Figure 6.5 shows the model that achieved the highest classification accuracy of touch modalities. The model consists of two convolutional layers, two fully connected layers, and one max pooling layer with batch normalization applied in all layers. Each convolution layer consists of seven 3×3 kernels. The first fully connected layer consists of fifteen kernels. The input of the first and last layer is represented in 32-bit floating-point, while the input

of the remaining layers is binary. The weights of all layers are binarized while the network doesn't use any bias values in all layers.

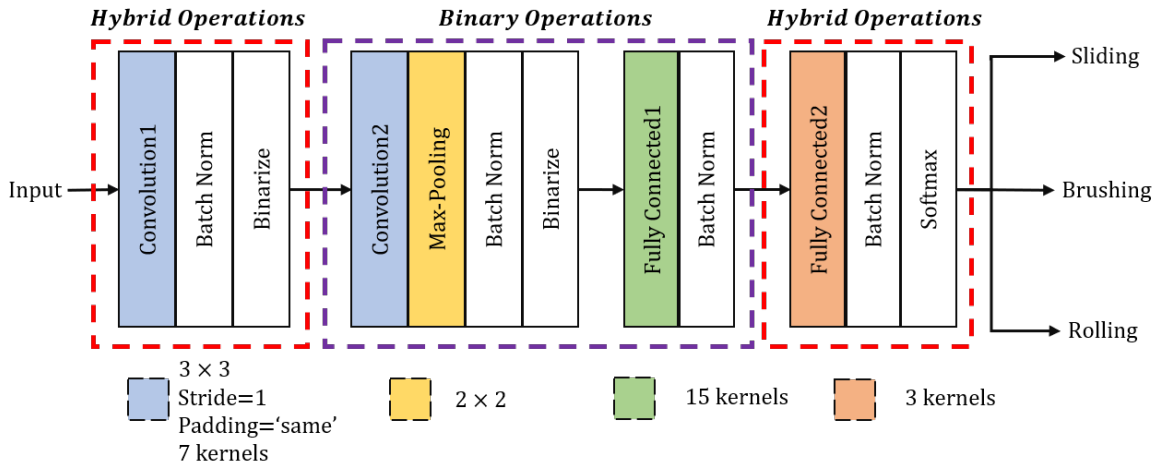


Fig. 6.5 H-BWN Best Model

Figure 6.6 reports the average classification accuracy among 10 runs for the binarized networks under batch-size=100 and number of epochs = 500. The obtained results show that BNN, BWN, and BAN were not able to achieve an accuracy above 50%. However, the proposed hybrid-precision methodology has significantly increased the accuracy reaching 71% for H-BWN. Such results account for a compensation of the precision loss due to the binarization of the network. The compensation is at the expense of increasing the memory storage requirements from 2.3 KB to 4.8 KB; however, such storage is available in mainstream microcontrollers (e.g. STM32F0x2 with 16 KB memory storage).

When compared to similar solutions for the same touch modality classification problem, the proposed H-BWN is superior in the aspects of accuracy, number of operations, and the trainable model parameters as shown in Table 6.2. H-BWN offers up to 99.9% reduction in the number of operations compared to SVM and DCNN algorithms [32]. Similarly, the H-BWN model size resembles a 97.8% and 99.9% compression rate compared to that of SVM and DCNN respectively. The achieved computational reduction and compression are accompanied with the highest classification accuracy of 77.88%.

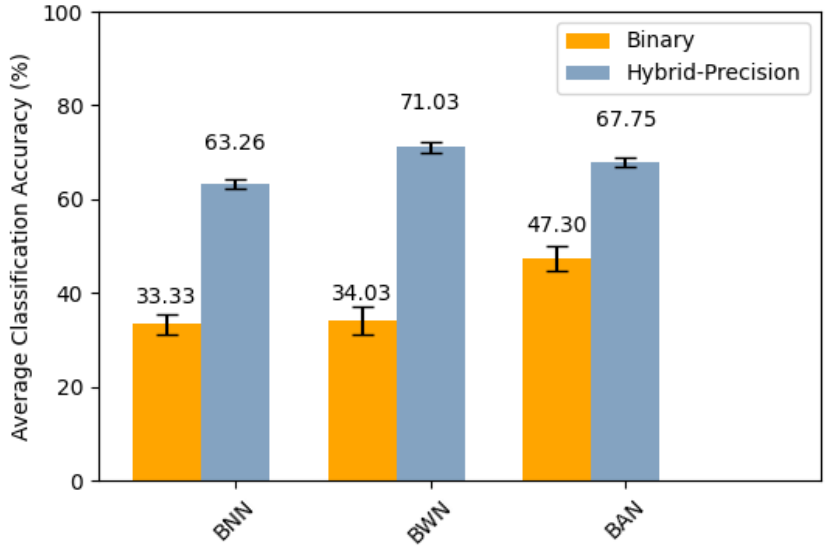


Fig. 6.6 Model Performance for Different Network Configurations

Table 6.2 Best H-BWN Performance in Comparison with the state of the art

	H-BWN	SVM [5]	DCNN [32]
Highest Classification Accuracy	77.88%	76.6%	76.9%
Number of Gops	3.2×10^{-5}	0.545	109×10^{-3}
Reduction in Gops	NA	99.99%	99.97%
Model Parameters	1.42K	67.2K	540M
Reduction in Model Parameters	NA	97.8%	99.9%

6.4 H-BWN Accelerator Design and Implementation

6.4.1 Accelerator Architecture

This section details the hardware architecture and implementation of the H-BWN presented in Figure 6.5. The proposed H-BWN accelerator architecture is shown in Figure 6.7. The architecture employs an array of processing elements (PEs) to offer configurable degree of parallelism for the convolution and matrix multiplication computations throughout different layers. Due to binarization of the kernels of all layers, these kernels can be stored efficiently using on-chip memory. Such design choice mitigates the latency overhead imposed by using external memory for kernel storage. The different units of the accelerator are:

CONV1 Unit: This unit computes the output of the first convolution layer. CONV1 consists of an array of Quantized Input, Binarized Kernel Convolution (QC) processing

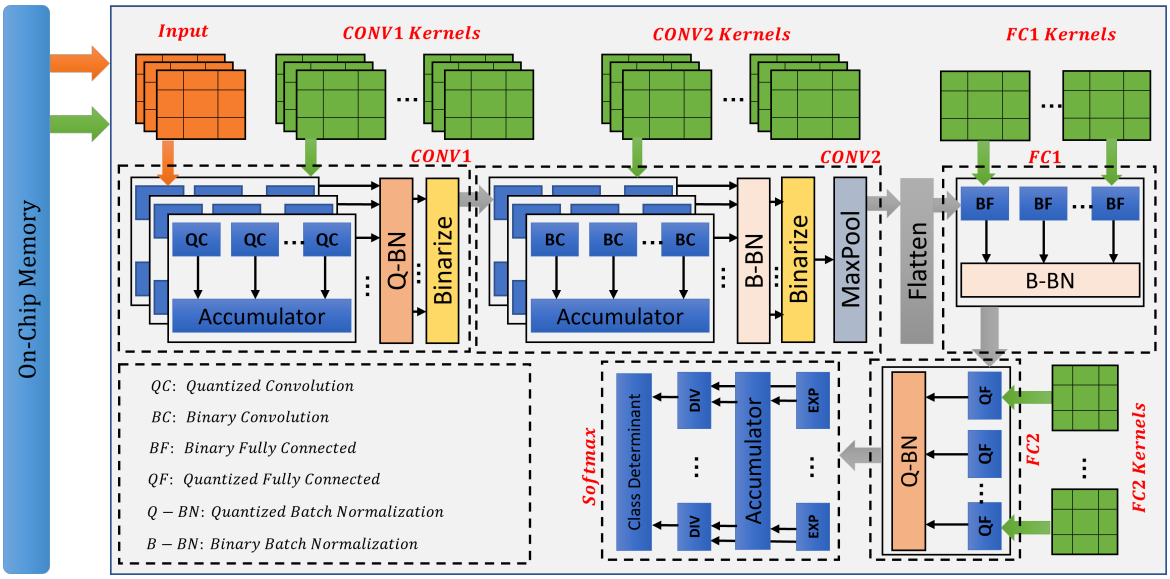


Fig. 6.7 Proposed H-CNN Hardware Accelerator Architecture

elements, a Quantized Batch Normalization (Q-BN) unit, and a Binarize process. Each QC computes the convolution between 3×3 input and kernel. The output of each 8 (i.e. input feature maps) QCs are accumulated to form one element of the $4 \times 4 \times 7$ output. For each output feature map, the computations of each 8 QCs are performed in parallel. As shown in Figure 6.8, the convolution between a quantized input and binary kernel can be computed using a set of bitwise-AND operations and an adder.

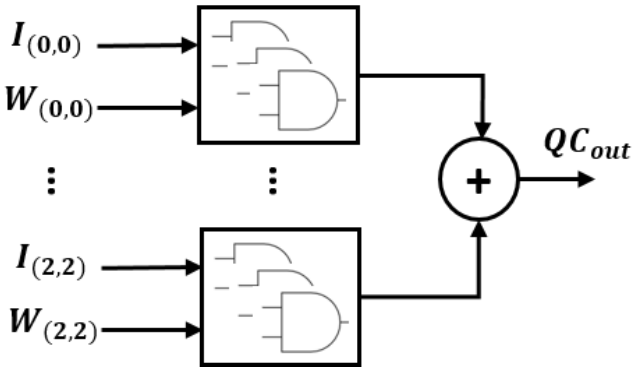


Fig. 6.8 Quantized Input, Binarized Kernel Convolution (QC) Processing Element

Then, batch normalization is to be applied on the $4 \times 4 \times 7$ convolution output according to the equation:

$$y_i = \gamma\left(\frac{x_i - \mu}{\sqrt{\sigma^2 + \varepsilon}}\right) + \beta \tag{6.10}$$

where x_i and y_i are the input and output of the batch-normalization respectively. μ, γ, σ, β, and ε are constants obtained from the training phase. To design a hardware friendly architecture of the Q-BN unit, (6.10) can be written as:

$$y_i = \frac{\gamma}{\sqrt{\sigma^2+\varepsilon}}(x_i - (\mu - \frac{\sqrt{\sigma^2+\varepsilon}}{\gamma}\beta)) \tag{6.11}$$

which in return can be simplified into the form:

$$y_i = \alpha_{i2}(x_i - \alpha_{i1}) \tag{6.12}$$

where $\alpha_{i1} = \mu - \frac{\sqrt{\sigma^2+\varepsilon}}{\gamma}\beta$ and $\alpha_{i2} = \frac{\gamma}{\sqrt{\sigma^2+\varepsilon}}$ are constants to be computed offline. Thus a Q-BN unit can be designed as shown in Figure 6.9 using a subtractor and a multiplier. Finally, the output of the CONV1 layer is obtained by binarizing the output of the Q-BN unit using the *sign* function.

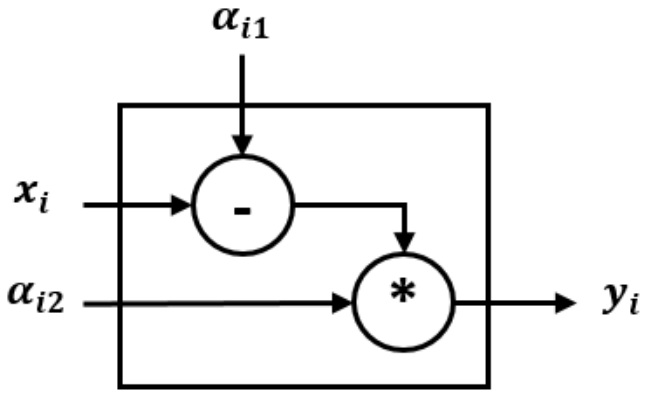


Fig. 6.9 Quantized Batch Normalization (Q-BN) Unit

CONV2 Unit: This unit computes the output of the second convolution layer. CONV2 consists of an array of Binarized Input, Binarized Kernel Convolution (BC) processing elements, a Binarized Batch Normalization (B-BN) unit, and a Binarize process followed by a Maxpool operation. According to 6.5, the convolution is computed using a set of XNOR gates followed by a popcount operation (i.e. number of 1's bits) as shown in Figure 6.10.

To apply batch normalization on the BC binary output, we adopted the *"Batch Normalization Free Binarized Artificial Neuron (AN)"* approach presented in [192]. The approach states that an artificial neuron with batch normalization can be converted into a binarized AN with an integer bias W', where W' is computed after training using the batch normalization constants. The integer output of the B-BN unit is binarized using the *sign* function. To reduce the hardware complexity of CONV2 unit, max pooling is applied at the end compared to the

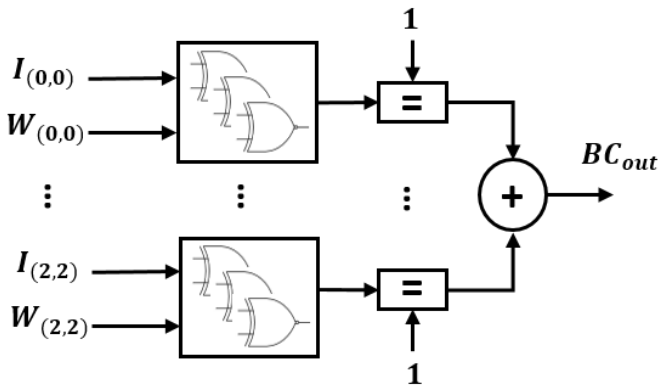


Fig. 6.10 Binarized Input, Binarized Kernel Convolution (BC) Processing Element

model shown in Figure 6.5. This is due to the fact the output of the *sign* function is binary compared to the integer output of the B-BN unit. Hence, MaxPool block can be designed using OR gates as shown in Figure 6.11.

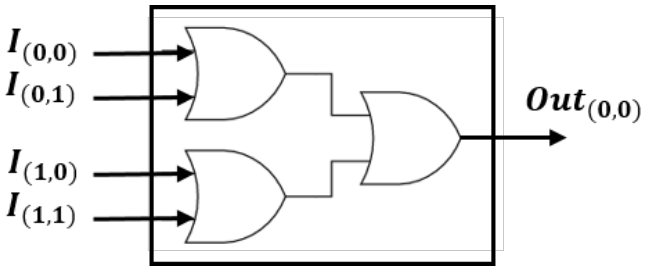


Fig. 6.11 Maxpool Operation on Binary Inputs

FC1 Unit: This unit computes the vector-matrix multiplication of binary input of size 28 (i.e the $2 \times 2 \times 7$ output of CONV2 is flattened to a vector fo size 28) and binary kernel of size 28×15. FC1 unit consists of 15 Binary Fully Connected (BF) processing elements that utilize the same design shown in Figure 6.10, which uses a set of AND gates and an adder to compute the multiplication of binary data. Similarly, batch normalization on binary values is performed using a B-BN unit. The output of the FC1 unit is represented as a vector consisting of 15 (i.e $28 \times (28 \times 15) = 1 \times 15$) quantizied elements where all the elements are obtained at once by operating the PEs in parallel.

FC2 Unit: This unit computes the vector-matrix multiplication of quantized input of size 15 and binary kernel of size 15×3. FC2 unit consists of 15 Quantized Fully Connected (QF) processing elements that utilize the same design shown in Figure 6.8, which uses a set of XNOR gates and a popcount to compute the multiplication of quantized input and binary kernel. Then, batch normalization is applied using a Q-BN unit. The output of the FC2 unit is a vector that consists of 3 quantized elements (i.e $15 \times (15 \times 3) = 1 \times 3$).

Softmax Unit: This unit assigns a class to the tactile input based on the highest value of $f(x_i)$ defined as:

$$f(x_i) = \frac{e^{x_i}}{\sum_{n=1}^{3} e^{x_i}} \tag{6.13}$$

The Softmax unit contains three main sub-units: (1) Exponent (Exp), which is used to compute the term e^{x_i}, (2) Division (DIV), which is used to compute $f(x_i)$, and (3) Class Determinant which is used to assign a class for the input.

–Exponent sub-unit:

The exponent function can be written as:

$$e^x = 2^{xlog(e)} \tag{6.14}$$

Let the constant $log(e)$ defined as y, then (6.14) can be written as:

$$e^x = 2^{x*y} \tag{6.15}$$

Thus, to compute the exponent of x, a multiplication of x and y is performed, followed by a base 2 exponentiation. Assume that $x*y = z$, then the term 2^{x*y} can be represented as 2^z. Since x and $loge$ are quantized values, their multiplication z is composed of an integer part i and a fractional part f. Hence, (6.15) can be simplified as:

$$e^x = 2^{i+f} = 2^i * 2^f \tag{6.16}$$

The term 2^i can be computed using left shift operations only. As for the term 2^f, we adopted the use of linear approximation to obtain an efficient hardware design. Suppose that the term 2^f can be approximated as:

$$2^f = K * f + b \tag{6.17}$$

Through simulations, several values of k and b can be determined based on the desired accuracy range of 2^f. Figure 6.12 shows the linear approximation function adopted for the computation of the term 2^f where the values of k and b are set to 0.724 and 1.008 respectively. Following the plot in Figure 6.12, it can be seen that for values of $f > 4$, the approximation error starts to increase rapidly. However, the fractional part f is bounded between -0.99 and 0.99, thus for the presented architecture in Figure 6.13, the linear approximation leads to an error in the order of 10^{-6}. Such architecture is capable to compute the exponent using multiplication, addition, and shift operations.

–Division sub-unit:

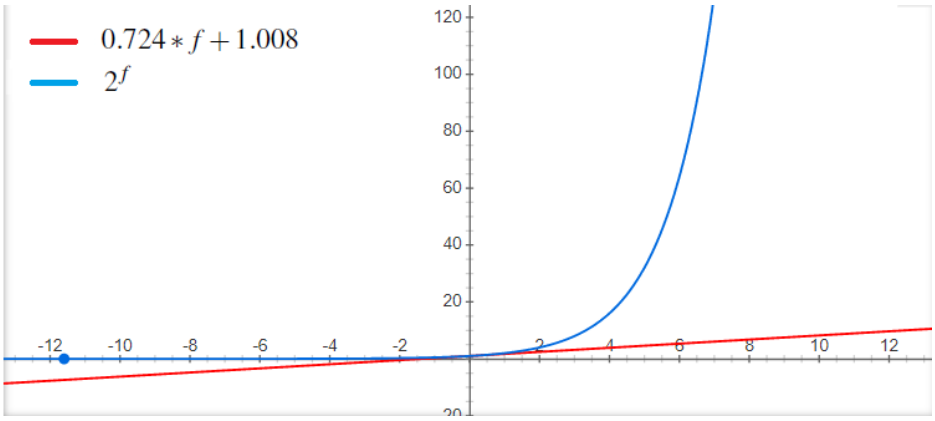


Fig. 6.12 Plot of the functions 2^f and $0.724 * f + 1.008$

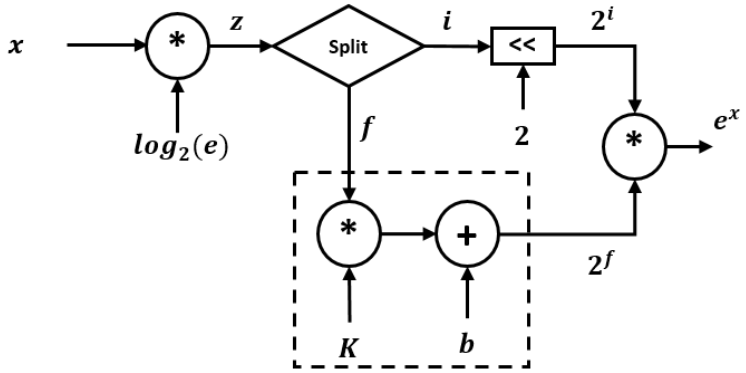


Fig. 6.13 Exponent Sub-unit Design using Linear Approximation

The division operation can be computed using shift and subtract operations based on the non-restoring division algorithm [193]. Figure 6.14 shows a flowchart that highlights the basic operations of the non-restoring division method. To operate on fixed-point data, the $Q[0] = 0$ and $Q[0] = 1$ steps for integers in the non-restoring division algorithm has been changed to $Q[8] = 0$ and $Q[8] = 1$ respectively for the fixed-point representation with $< 8,8 >$ precision.

Figure 6.15 shows the proposed architecture for the division sub-unit where $N = \sum_{n=1}^{3} e^{x_i}$, $M = e^{x_i}$, *Rem* is the remainder, and *Qout* is the quotient. he rational behind computing the reciprocal of the Softmax function is the use of the non-restoring division algorithm such that the dividend is greater than the divisor. The architecture involves addition, subtraction, and shift operations. Also, multiplexers are used for conditional testing and registers are utilized for intermediate results.

–*Class Determinant sub-unit*:

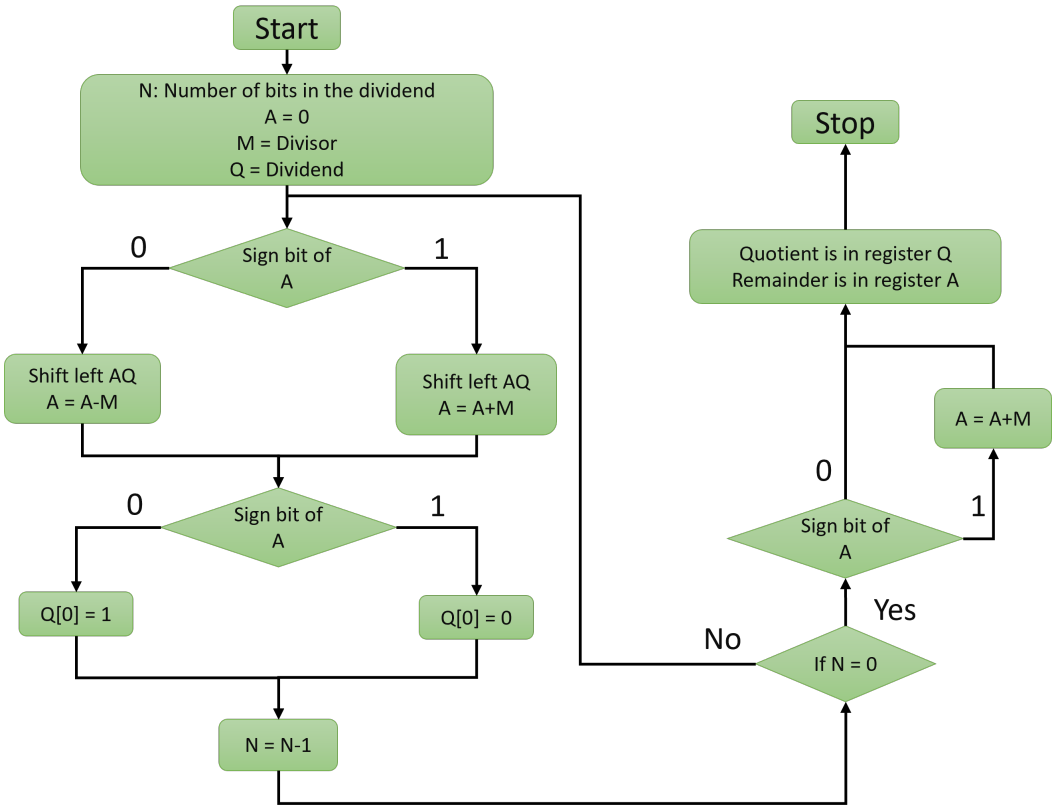


Fig. 6.14 Non-Restoring Division Algorithm

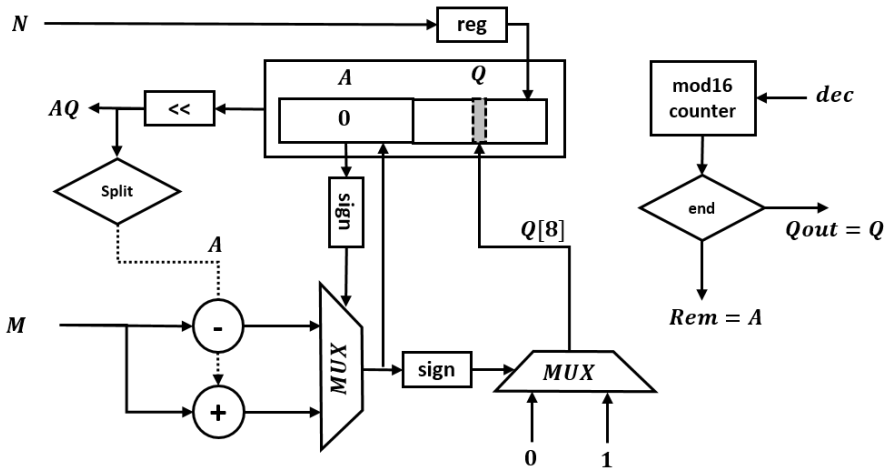


Fig. 6.15 Division Sub-unit Design based on Non-Restoring Division Algorithm

After computing the value of $f(x_i)$ for the 3 classes, the class determinant sub-unit assigns a modality to the input based on the minimum value of $f(x_i)^{-1}$ as the reciprocal of the Softmax is computed.

6.4.2 Accelerator Implementation

The proposed accelerator architecture shown in Figure 6.7 has been designed with 16-bit fixed-point representation and $< 8,8 >$ (sign bit, 7-bits for integer part, and 8-bits for fractional part) precision in C++ using Vivado HLS. Targeting the Zynqberry platform operating at 100 MHz, the accelerator has been exported as an Intellectual Property (IP) to Vivado. In Vivado, behavioral, then post-implementation functional and timing simulations are performed to obtain the required resources and time latency. As for power consumption, it is recorded through vector-based approach using a *saif* file generated after implementation.

Table 6.3 shows the implementation report of the H-BWN accelerator. The reported resources show that the H-BWN is adequate for embedded implementations on resource-limited platforms with an average utilization percentage of 19%. The large number of DSPs is due to the level of parallelization introduced in the architecture to insure real-time functionality. This is evident in the time latency of 0.8 ms with a power consumption of 53 mW which verifies the real-time requirements of touch classification [4].

Table 6.3 H-BWN Implementation Results Targeting Zynqberry Platform

Resource	**Utilization (%)**
LUT	3886 (22%)
FF	3365 (9.5%)
DSP	48 (60%)
BRAM	4 (3.3%)
Time Latency (ms)	0.8
PowerConsumption (mW)	53

Targeting the touch modality classification problem introduced in section 6.3.1, up to the date of writing this dissertation, this is the only work that reports the hardware implementation of a Quantized/Binary neural network. Thus, a comparison with existing solutions mainly Support Vector Machine (SVM) is performed in terms of "energy per classification" computed as $E = P \times T$, where P is the dynamic power consumption and T is the time latency. For this purpose, three implementations are considered:

- SVM implementation on Parallel Ultra-Low Platform (PULP) running on 1 core.
- SVM implementation on PULP running on 2 cores.
- SVM implementation on Xilinx Zedboard.

Table 6.4 shows the comparison with similar solutions. With a 77% accuracy, the proposed H-BWN offers a 6875$\times$ and 4125$\times$ speedup compared to the SVM implementation

Table 6.4 H-BWN Performance Comparison with Similar Solutions

Algorithm	**H-BWN**	**SVM [1]**	**SVM [1]**	**SVM [2]**
Classification Accuracy (%)	77	72.8	72.8	70.9
Device	Zynqberry	PULP: 1 Core	PULP: 2 Cores	ZedBoard
Power Consumption (mW)	53	21	28	NA
Time Latency (ms)	0.8	3.3×10^3	5.5×10^3	28
Energy per classification (mJ)	42.4×10^{-3}	69.3	154	NA

on PULP [56] with 1 core and 2 cores respectively. Although the authors in [56] have reported a lower power consumption of 21 mW and 28 mW for 1 core and 2 cores implementation respectively, the significant speedup achieved by the proposed H-BWN leads to an energy reduction up to 99.6%. Compared to the SVM implementation on ZedBoard in [71], a 35$\times$ speedup has been recorded. Althought no power consumption details are provided in [71], it is expected that the proposed accelerator is more power efficient than the SVM implementation on Zedboard due to the massive speedup and ultra low power consumption of the proposed accelerator.

6.5 Conclusion

This chapter presents a Binary Convolution Neural Network architecture based on hybrid precision approach. First, a design methodology is provided starting from selecting the network topology, layers' characteristics, quantization level and training strategy. The H-BWN architecture consists of two convolution and two fully connected layers; the first and last layers use 32-bit floating-point activations and binary weights while the hidden layers are completely binarized. H-BWN achieves a classification accuracy boost more than 35% against traditional BNNs when validated on a touch modalities problem. Compared to SVM and DCNN algorithms targeting the same classification problem, the H-BWN provides higher classification accuracy of 77% accompanied with a 99% reduction in the number of operations and model size. When implemented on Zynqberry platform, the H-BWN accelerator provides real-time classification within 0.8 ms while consuming 53 mW. Such performance offers an energy reductions up to 99% compared to SVM accelerators implemented on PULP and Zedboard platforms.

Chapter 7

Conclusion

The deployment of machine learning algorithms on embedded devices for applications with a constrained requirements is an active challenge. In this dissertation, we investigated two of the main methods that are widely used to reduce the obstacles faced while overcoming such challenge. The first method is the use of approximate computing to reduce the computational complexity of ML algorithms with an acceptable margin of error. The second method is to design custom hardware accelerator architectures optimized for a certain algorithm implemented on a specific hardware platform.

For a tactile data processing application, three novel hardware accelerators are designed with both exact and approximate computations. A selection-based sorter that selects the k smallest numbers out of a sequence without the need to sort the complete sequence is proposed. When embedded in an exact kNN accelerator, a real-time classification of touch modalities is achieved while consuming $6\mu J$ on Zynqberry. Compared to similar kNN architectures, the proposed kNN achieves a speedup between $1.4\times$ and $875\times$ with 41% to 94% less energy consumption and 12% to 94% average hardware area reduction. As for approximate kNN, a 56.4% average area reduction, a speedup by $2.3\times$, and an energy reduction of about 69% are recorded compared to its exact counterpart. A shallow neural network is designed to predict the singular vectors. The network is used to replace the traditional one-sided Jacobi algorithm used as an SVD computation block of the tensorial SVM. Such design methodology provides a classification speedup up to $131\times$ with a 39% and 50% resources and power reductions respectively compared to similar stat-of-the-art solution. The noticeable performance improvements pave the way towards the deployment of intelligence on resource-limited devices for power-constrained applications (e.g. prosthetic). To combat the accuracy drop in binary neural networks, and benefit from the high performance of CNNs, a hybrid precision binary weight neural network is designed. A 35% accuracy increase in classifying touch modalities compared to traditional BCNN topology is recorded.

A real-time classification within 0.8 ms with a $42.4\mu J$ energy per classification could be achieved for FPGA implementation. Compared to exiting solutions, an energy reduction up to 99% accompanied with a speedup up to $6875\times$ is provided.

With the continuous adoption of machine learning algorithms in different domains, our future work involves tracking the advancements of such algorithms especially deep learning. For instance, Transformers have been lately investigated for the use of image classification [194], [195]. Which could be possibly adopted for tactile data processing due to the tensorial nature of tactile signals. Consequently, monitoring the new methodologies for the design and implementation of hardware accelerators. In addition, due to the error resilience nature of ML algorithms, new approximate computing techniques are emerging such as adaptive approximate computing [196] and approximate adders with Single LUT delay [115].

References

[1] M. S. Mahmud, J. Z. Huang, S. Salloum, T. Z. Emara, and K. Sadatdiynov, "A survey of data partitioning and sampling methods to support big data analysis," *Big Data Min. Anal.*, vol. 3, pp. 85–101, June 2020.

[2] D. Tribe, "Flexibility can help car makers cope with chip supply challenge," *Engineering & Technology*, vol. 16, pp. 19–19, Mar. 2021.

[3] S. Venkataramani, S. T. Chakradhar, K. Roy, and A. Raghunathan, "Approximate computing and the quest for computing efficiency," in *Proceedings of the 52nd Annual Design Automation Conference on - DAC '15*, (San Francisco, California), pp. 1–6, ACM Press, 2015.

[4] P. P. Lele, D. C. Sinclair, and G. Weddell, "The reaction time to touch," *The Journal of Physiology*, vol. 123, pp. 187–203, Jan. 1954.

[5] P. Gastaldo, L. Pinna, L. Seminara, M. Valle, and R. Zunino, "Computational Intelligence Techniques for Tactile Sensing Systems," *Sensors*, vol. 14, pp. 10952–10976, June 2014.

[6] "Apple unleashes M1."

[7] D. Fujiki, X. Wang, A. Subramaniyan, and R. Das, "In-/Near-Memory Computing," *Synthesis Lectures on Computer Architecture*, vol. 16, pp. 1–140, Aug. 2021.

[8] R. Dahiya, G. Metta, M. Valle, and G. Sandini, "Tactile Sensing—From Humans to Humanoids," *IEEE Trans. Robot.*, vol. 26, pp. 1–20, Feb. 2010.

[9] S. Wold, K. Esbensen, and P. Geladi, "Principal component analysis," *Chemometrics and intelligent laboratory systems*, vol. 2, no. 1-3, pp. 37–52, 1987.

[10] L. Derksen, *Visualising high-dimensional datasets using PCA and t-SNE in Python*. Apr. 2019.

[11] H. Hu, Y. Han, A. Song, S. Chen, C. Wang, and Z. Wang, "A Finger-Shaped Tactile Sensor for Fabric Surfaces Evaluation by 2-Dimensional Active Sliding Touch," *Sensors*, vol. 14, pp. 4899–4913, Mar. 2014.

[12] Y.-H. Liu, Y.-T. Hsiao, W.-T. Cheng, Y.-C. Liu, and J.-Y. Su, "Low-Resolution Tactile Image Recognition for Automated Robotic Assembly Using Kernel PCA-Based Feature Fusion and Multiple Kernel Learning-Based Support Vector Machine," *Mathematical Problems in Engineering*, vol. 2014, pp. 1–11, 2014.

[13] B. Schölkopf, A. Smola, and K.-R. Müller, "Kernel principal component analysis," in *International conference on artificial neural networks*, pp. 583–588, Springer, 1997.

[14] M. Schopfer, H. Ritter, and G. Heidemann, "Acquisition and Application of a Tactile Database," in *Proceedings 2007 IEEE International Conference on Robotics and Automation*, pp. 1517–1522, Apr. 2007.

[15] A. Weingessel and K. Hornik, "Local PCA algorithms," *IEEE Transactions on Neural Networks*, vol. 11, pp. 1242–1250, Nov. 2000.

[16] P. Comon, "Independent component analysis, A new concept?," *Signal Processing*, vol. 36, pp. 287–314, Apr. 1994.

[17] S. Haykin and Z. Chen, "The cocktail party problem," *Neural computation*, vol. 17, no. 9, pp. 1875–1902, 2005.

[18] K. Lee, T. Ikeda, T. Miyashita, H. Ishiguro, and N. Hagita, "Separation of tactile information from multiple sources based on spatial ICA and time series clustering," in *2011 IEEE/SICE International Symposium on System Integration (SII)*, (Kyoto, Japan), pp. 791–796, IEEE, Dec. 2011.

[19] P. Xanthopoulos, P. M. Pardalos, and T. B. Trafalis, "Linear Discriminant Analysis," in *Robust Data Mining* (P. Xanthopoulos, P. M. Pardalos, and T. B. Trafalis, eds.), SpringerBriefs in Optimization, pp. 27–33, New York, NY: Springer, 2013.

[20] M. Pal, A. Khasnobish, A. Konar, D. N. Tibarewala, and R. Janarthanan, "Classification of deformable and non-deformable surfaces by tactile image analysis," in *Proceedings of The 2014 International Conference on Control, Instrumentation, Energy and Communication (CIEC)*, (Calcutta, India), pp. 626–630, IEEE, Jan. 2014.

[21] H. Nguyen, L. Osborn, M. Iskarous, C. Shallal, C. Hunt, J. Betthauser, and N. Thakor, "Dynamic Texture Decoding Using a Neuromorphic Multilayer Tactile Sensor," in *2018 IEEE Biomedical Circuits and Systems Conference (BioCAS)*, (Cleveland, OH), pp. 1–4, IEEE, Oct. 2018.

[22] W. K, "Tactile sensing for ground classification," *Journal of Automation Mobile Robotics and Intelligent Systems*, vol. 7, no. 2, pp. 18–23, 2013.

[23] I. Bandyopadhyaya, D. Babu, A. Kumar, and J. Roychowdhury, "Tactile sensing based softness classification using machine learning," in *2014 IEEE International Advance Computing Conference (IACC)*, (Gurgaon, India), pp. 1231–1236, IEEE, Feb. 2014.

[24] Z. Yi, Y. Zhang, and J. Peters, "Bioinspired tactile sensor for surface roughness discrimination," *Sensors and Actuators A: Physical*, vol. 255, pp. 46–53, Mar. 2017.

[25] T. Bhattacharjee, J. M. Rehg, and C. C. Kemp, "Haptic classification and recognition of objects using a tactile sensing forearm," in *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*, (Vilamoura-Algarve, Portugal), pp. 4090–4097, IEEE, Oct. 2012.

[26] Y. Gao, L. A. Hendricks, K. J. Kuchenbecker, and T. Darrell, "Deep learning for tactile understanding from visual and haptic data," in *2016 IEEE International Conference on Robotics and Automation (ICRA)*, (Stockholm, Sweden), pp. 536–543, IEEE, May 2016.

[27] M. Kaboli, P. Mittendorfer, V. Hugel, and G. Cheng, "Humanoids learn object properties from robust tactile feature descriptors via multi-modal artificial skin," in *2014 IEEE-RAS International Conference on Humanoid Robots*, (Madrid, Spain), pp. 187–192, IEEE, Nov. 2014.

[28] M. Kaboli, R. Walker, and G. Cheng, "Re-using prior tactile experience by robotic hands to discriminate in-hand objects via texture properties," in *2016 IEEE International Conference on Robotics and Automation (ICRA)*, (Stockholm, Sweden), pp. 2242–2247, IEEE, May 2016.

[29] P. Gastaldo, L. Pinna, L. Seminara, M. Valle, and R. Zunino, "A Tensor-Based Pattern-Recognition Framework for the Interpretation of Touch Modality in Artificial Skin Systems," *IEEE Sensors Journal*, vol. 14, pp. 2216–2225, July 2014.

[30] D. Xu, G. E. Loeb, and J. A. Fishel, "Tactile identification of objects using Bayesian exploration," in *2013 IEEE International Conference on Robotics and Automation*, (Karlsruhe, Germany), pp. 3056–3061, IEEE, May 2013.

[31] M. Kaboli, A. Long, and G. Cheng, "Humanoids learn touch modalities identification via multi-modal robotic skin and robust tactile descriptors," *Advanced Robotics*, vol. 29, pp. 1411–1425, Nov. 2015.

[32] M. Alameh, A. Ibrahim, M. Valle, and G. Moser, "DCNN for Tactile Sensory Data Classification based on Transfer Learning," in *2019 15th Conference on Ph.D Research in Microelectronics and Electronics (PRIME)*, (Lausanne, Switzerland), pp. 237–240, IEEE, July 2019.

[33] Shan Luo, Wenxuan Mou, Min Li, K. Althoefer, and Hongbin Liu, "Rotation and translation invariant object recognition with a tactile sensor," in *IEEE SENSORS 2014 Proceedings*, (Valencia), pp. 1030–1033, IEEE, Nov. 2014.

[34] J. Schill, J. Laaksonen, M. Przybylski, V. Kyrki, T. Asfour, and R. Dillmann, "Learning continuous grasp stability for a humanoid robot hand based on tactile sensing," in *2012 4th IEEE RAS & EMBS International Conference on Biomedical Robotics and Biomechatronics (BioRob)*, (Rome, Italy), pp. 1901–1906, IEEE, June 2012.

[35] F. Veiga, H. van Hoof, J. Peters, and T. Hermans, "Stabilizing novel objects by learning to predict tactile slip," in *2015 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, (Hamburg, Germany), pp. 5065–5072, IEEE, Sept. 2015.

[36] Z. Su, K. Hausman, Y. Chebotar, A. Molchanov, G. E. Loeb, G. S. Sukhatme, and S. Schaal, "Force estimation and slip detection/classification for grip control using a biomimetic tactile sensor," in *2015 IEEE-RAS 15th International Conference on Humanoid Robots (Humanoids)*, (Seoul, South Korea), pp. 297–303, IEEE, Nov. 2015.

[37] A. Dabbous, M. Mastella, A. Natarajan, E. Chicca, M. Valle, and C. Bartolozzi., "Artificial Bio-Inspired Tactile Receptive Fields for Edge Orientation Classification," in *2021 IEEE International Symposium on Circuits and Systems (ISCAS)*, (Daegu, Korea), pp. 1–5, IEEE, May 2021.

[38] J. A. Pruszynski and R. S. Johansson, "Edge-orientation processing in first-order tactile neurons," *Nat Neurosci*, vol. 17, pp. 1404–1409, Oct. 2014.

[39] M. Malviya and K. Desai, "Build Orientation Optimization for Strength Enhancement of FDM Parts Using Machine Learning based Algorithm," *CADandA*, vol. 17, pp. 783–796, Nov. 2019.

[40] *Big-O Algorithm Complexity Cheat Sheet (Know Thy Complexities!) @ericdrowell.*

[41] K. He and J. Sun, "Convolutional Neural Networks at Constrained Time Cost," *arXiv:1412.1710 [cs]*, Dec. 2014. arXiv: 1412.1710.

[42] A. Canziani, A. Paszke, and E. Culurciello, "An Analysis of Deep Neural Network Models for Practical Applications," *arXiv:1605.07678 [cs]*, Apr. 2017. arXiv: 1605.07678.

[43] X. Song, T. Xie, and S. Fischer, "A Memory-Access-Efficient Adaptive Implementation of kNN on FPGA through HLS," in *2019 IEEE 37th International Conference on Computer Design (ICCD)*, (Abu Dhabi, United Arab Emirates), pp. 177–180, IEEE, Nov. 2019.

[44] J. Vieira, R. P. Duarte, and H. C. Neto, "kNN-STUFF: kNN STreaming Unit for Fpgas," *IEEE Access*, vol. 7, pp. 170864–170877, 2019.

[45] H. Hussain, K. Benkrid, C. Hong, and H. Seker, "An adaptive FPGA implementation of multi-core K-nearest neighbour ensemble classifier using dynamic partial reconfiguration," in *22nd International Conference on Field Programmable Logic and Applications (FPL)*, (Oslo, Norway), pp. 627–630, IEEE, Aug. 2012.

[46] K. R. Townsend, S. Sun, T. Johnson, O. G. Attia, P. H. Jones, and J. Zambreno, "k-NN text classification using an FPGA-based sparse matrix vector multiplication accelerator," in *2015 IEEE International Conference on Electro/Information Technology (EIT)*, (Dekalb, IL, USA), pp. 257–263, IEEE, May 2015.

[47] Y. Pu, J. Peng, L. Huang, and J. Chen, "An Efficient KNN Algorithm Implemented on FPGA Based Heterogeneous Computing System Using OpenCL," in *2015 IEEE 23rd Annual International Symposium on Field-Programmable Custom Computing Machines*, (Vancouver, BC, Canada), pp. 167–170, IEEE, May 2015.

[48] A. Al-Zoubi, K. Tatas, and C. Kyriacou, "Design space exploration of the KNN imputation on FPGA," in *2018 7th International Conference on Modern Circuits and Systems Technologies (MOCAST)*, (Thessaloniki), pp. 1–4, IEEE, May 2018.

[49] E. S. Manolakos and I. Stamoulias, "IP-cores design for the kNN classifier," in *Proceedings of 2010 IEEE International Symposium on Circuits and Systems*, (Paris, France), pp. 4133–4136, IEEE, May 2010.

[50] C. Cedeno Z., J. Cordova-Garcia, V. Asanza A., R. Ponguillo, and L. Munoz M., "k-NN-Based EMG Recognition for Gestures Communication with Limited Hardware Resources," in *2019 IEEE SmartWorld, Ubiquitous Intelligence & Computing, Advanced & Trusted Computing, Scalable Computing & Communications, Cloud & Big Data Computing, Internet of People and Smart City Innovation (SmartWorld/SCALCOM/UIC/ATC/CBDCom/IOP/SCI)*, (Leicester, United Kingdom), pp. 812–817, IEEE, Aug. 2019.

[51] Miren Tian, Xin'an Wang, Xing Zhang, Zhiqiang Yang, Jipan Huang, and Hao Chen, "The implementation of a KNN classifier on FPGA with a parallel and pipelined architecture based on Predetermined Range Search," in *2016 13th IEEE International Conference on Solid-State and Integrated Circuit Technology (ICSICT)*, (Hangzhou, China), pp. 1491–1493, IEEE, Oct. 2016.

[52] H. Peng, L. Huang, and J. Chen, "An efficient FPGA implementation for odd-even sort based KNN algorithm using OpenCL," in *2016 International SoC Design Conference (ISOCC)*, (Jeju, South Korea), pp. 207–208, IEEE, Oct. 2016.

[53] Shereen Moataz Afifi, Hamid GholamHosseini and Roopak Sinha, "Hardware Implementations of SVM on FPGA: AState-of-the-Art Review ofCurrent Practice," vol. 2, pp. 2348–7968, Nov. 2015.

[54] M. Signoretto, L. De Lathauwer, and J. A. Suykens, "A kernel-based framework to tensorial data analysis," *Neural Networks*, vol. 24, pp. 861–874, Oct. 2011.

[55] Z. Yi, T. Xu, W. Shang, and X. Wu, "Touch Modality Identification With Tensorial Tactile Signals: A Kernel-Based Approach," *IEEE Trans. Automat. Sci. Eng.*, pp. 1–10, 2021.

[56] M. Osta, A. Ibrahim, M. Magno, M. Eggimann, A. Pullini, P. Gastaldo, and M. Valle, "An Energy Efficient System for Touch Modality Classification in Electronic Skin Applications," in *2019 IEEE International Symposium on Circuits and Systems (ISCAS)*, (Sapporo, Japan), pp. 1–4, IEEE, May 2019.

[57] R. A. Patil, G. Gupta, V. Sahula, and A. Mandal, "Power Aware Hardware Prototyping of Multiclass SVM Classifier Through Reconfiguration," in *2012 25th International Conference on VLSI Design*, (Hyderabad, India), pp. 62–67, IEEE, Jan. 2012.

[58] H. M. Hussain, K. Benkrid, and H. Seker, "Reconfiguration-based implementation of SVM classifier on FPGA for Classifying Microarray data," in *2013 35th Annual International Conference of the IEEE Engineering in Medicine and Biology Society (EMBC)*, (Osaka), pp. 3058–3061, IEEE, July 2013.

[59] H. M. Hussain, K. Benkrid, and H. Seker, "Dynamic partial reconfiguration implementation of the SVM/KNN multi-classifier on FPGA for bioinformatics application," in *2015 37th Annual International Conference of the IEEE Engineering in Medicine and Biology Society (EMBC)*, (Milan), pp. 7667–7670, IEEE, Aug. 2015.

[60] A. Luo, F. An, X. Zhang, and H. J. Mattausch, "A Hardware-Efficient Recognition Accelerator Using Haar-Like Feature and SVM Classifier," *IEEE Access*, vol. 7, pp. 14472–14487, 2019.

[61] Jesús Gimeno Sarciada, Horacio Lamel Rivera, and Matías Jiménez, "CORDIC algorithms for SVM FPGA implementation," vol. 7703, Apr. 2010.

[62] A.-H. M. Jallad and L. B. Mohammed, "Hardware Support Vector Machine (SVM) for satellite on-board applications," in *2014 NASA/ESA Conference on Adaptive Hardware and Systems (AHS)*, (Leicester, United Kingdom), pp. 256–261, IEEE, July 2014.

[63] V. S. Vranjković, R. J. R. Struharik, and L. A. Novak, "Reconfigurable Hardware for Machine Learning Applications," *Journal of Circuits, Systems and Computers*, vol. 24, p. 1550064, June 2015.

[64] S.-J. Kim, S.-Y. Lee, and K.-S. Cho, "Design of High-Performance Unified Circuit for Linear and Non-Linear SVM Classifications," *JSTS:Journal of Semiconductor Technology and Science*, vol. 12, pp. 162–167, June 2012.

[65] V. Vranjkovic and R. Struharik, "New architecture for SVM classifier and its application to telecommunication problems," in *2011 19thTelecommunications Forum (TELFOR) Proceedings of Papers*, (Belgrade, Serbia), pp. 1543–1545, IEEE, Nov. 2011.

[66] D. Mahmoodi, A. Soleimani, H. Khosravi, and M. Taghizadeh, "FPGA Simulation of Linear and Nonlinear Support Vector Machine," *JSEA*, vol. 04, no. 05, pp. 320–328, 2011.

[67] M. Cutajar, E. Gatt, I. Grech, O. Casha, and J. Micallef, "Hardware-based support vector machine for phoneme classification," in *Eurocon 2013*, (Zagreb, Croatia), pp. 1701–1708, IEEE, July 2013.

[68] M. Papadonikolakis and C.-S. Bouganis, "A novel FPGA-based SVM classifier," in *2010 International Conference on Field-Programmable Technology*, (Beijing, China), pp. 283–286, IEEE, Dec. 2010.

[69] L. A. Martins, G. A. M. Sborz, F. Viel, and C. A. Zeferino, "An SVM-based hardware accelerator for onboard classification of hyperspectral images," in *Proceedings of the 32nd Symposium on Integrated Circuits and Systems Design - SBCCI '19*, (São Paulo, Brazil), pp. 1–6, ACM Press, 2019.

[70] C. Kyrkou, T. Theocharides, and C.-S. Bouganis, "An embedded hardware-efficient architecture for real-time cascade Support Vector Machine classification," in *2013 International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS)*, (Agios konstantinos, Samos Island, Greece), pp. 129–136, IEEE, July 2013.

[71] M. Saleh, A. Ibrahim, F. Menichelli, Y. Mohanna, and M. Valle, "Efficient Machine Learning Algorithm for Embedded Tactile Data Processing," in *2021 IEEE International Symposium on Circuits and Systems (ISCAS)*, (Daegu, Korea), pp. 1–5, IEEE, May 2021.

[72] M. Magno, A. Ibrahim, A. Pullini, M. Valle, and L. Benini, "An Energy Efficient E-Skin Embedded System for Real-Time Tactile Data Decoding," *Journal of Low Power Electronics*, vol. 14, pp. 101–109, Mar. 2018.

[73] A. Ibrahim and M. Valle, "Real-Time Embedded Machine Learning for Tensorial Tactile Data Processing," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 65, pp. 3897–3906, Nov. 2018.

[74] M. Magno, A. Ibrahim, A. Pullini, M. Valle, and L. Benini, "Energy Efficient System for Tactile Data Decoding Using an Ultra-Low Power Parallel Platform," in *2017 New Generation of CAS (NGCAS)*, (Genova, Italy), pp. 17–20, IEEE, Sept. 2017.

[75] A. Ibrahim, P. Gastaldo, H. Chible, and M. Valle, "Real-Time Digital Signal Processing Based on FPGAs for Electronic Skin Implementation †," *Sensors*, vol. 17, p. 558, Mar. 2017.

[76] H. Qin, R. Gong, X. Liu, X. Bai, J. Song, and N. Sebe, "Binary neural networks: A survey," *Pattern Recognition*, vol. 105, p. 107281, Sept. 2020.

[77] M. Fischer and J. Wassner, "BinArray: A Scalable Hardware Accelerator for Binary Approximated CNNs," in *2021 IEEE 11th Annual Computing and Communication Workshop and Conference (CCWC)*, (NV, USA), pp. 0197–0205, IEEE, Jan. 2021.

[78] H. Peng, S. Zhou, S. Weitze, J. Li, S. Islam, T. Geng, A. Li, W. Zhang, M. Song, M. Xie, H. Liu, and C. Ding, "Binary Complex Neural Network Acceleration on FPGA : (Invited Paper)," in *2021 IEEE 32nd International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, (NJ, USA), pp. 85–92, IEEE, July 2021.

[79] B. S. Ajay and M. Rao, "Binary neural network based real time emotion detection on an edge computing device to detect passenger anomaly," in *2021 34th International Conference on VLSI Design and 2021 20th International Conference on Embedded Systems (VLSID)*, (Guwahati, India), pp. 175–180, IEEE, Feb. 2021.

[80] N. Fasfous, M.-R. Vemparala, A. Frickenstein, L. Frickenstein, and W. Stechele, "BinaryCoP: Binary Neural Network-based COVID-19 Face-Mask Wear and Positioning Predictor on Edge Devices," *arXiv:2102.03456 [cs]*, June 2021.

[81] A. Krizhevsky, "Learning Multiple Layers of Features from Tiny Images," 2009.

[82] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, "Imagenet: A large-scale hierarchical image database," in *2009 IEEE conference on computer vision and pattern recognition*, pp. 248–255, Ieee, 2009.

[83] M. Rastegari, V. Ordonez, J. Redmon, and A. Farhadi, "XNOR-Net: ImageNet Classification Using Binary Convolutional Neural Networks," in *Computer Vision – ECCV 2016* (B. Leibe, J. Matas, N. Sebe, and M. Welling, eds.), vol. 9908, pp. 525–542, Cham: Springer International Publishing, 2016.

[84] S. Zhou, Y. Wu, Z. Ni, X. Zhou, H. Wen, and Y. Zou, "DoReFa-Net: Training Low Bitwidth Convolutional Neural Networks with Low Bitwidth Gradients," *arXiv:1606.06160 [cs]*, Feb. 2018.

[85] M. Courbariaux, Y. Bengio, and J.-P. David, "BinaryConnect: Training Deep Neural Networks with Binary Weights during Propagations," in *Proceedings of the 28th International Conference on Neural Information Processing Systems - Volume 2*, NIPS'15, (Cambridge, MA, USA), pp. 3123–3131, MIT Press, 2015.

[86] X. Lin, C. Zhao, and W. Pan, "Towards Accurate Binary Convolutional Neural Network," *arXiv:1711.11294 [cs, stat]*, Nov. 2017.

[87] T.-Y. Lin, P. Goyal, R. Girshick, K. He, and P. Dollar, "Focal Loss for Dense Object Detection," in *2017 IEEE International Conference on Computer Vision (ICCV)*, (Venice), pp. 2999–3007, IEEE, Oct. 2017.

[88] L. Geiger and P. Team, "Larq: An Open-Source Library for Training Binarized Neural Networks," *JOSS*, vol. 5, p. 1746, Jan. 2020.

[89] J. Choi, Z. Wang, S. Venkataramani, P. I.-J. Chuang, V. Srinivasan, and K. Gopalakrishnan, "PACT: Parameterized Clipping Activation for Quantized Neural Networks," *arXiv:1805.06085 [cs]*, July 2018.

[90] I. Hubara, M. Courbariaux, D. Soudry, R. El-Yaniv, and Y. Bengio, "Binarized Neural Networks," in *Advances in Neural Information Processing Systems* (D. Lee, M. Sugiyama, U. Luxburg, I. Guyon, and R. Garnett, eds.), vol. 29, Curran Associates, Inc., 2016.

[91] S. Isobe and Y. Tomioka, "Low-bit Quantized CNN Acceleration based on Bit-serial Dot Product Unit with Zero-bit Skip," in *2020 Eighth International Symposium on Computing and Networking (CANDAR)*, (Naha, Japan), pp. 141–145, IEEE, Nov. 2020.

[92] S. Gupta, A. Agrawal, K. Gopalakrishnan, and P. Narayanan, "Deep Learning with Limited Numerical Precision," *arXiv:1502.02551 [cs, stat]*, Feb. 2015. arXiv: 1502.02551.

[93] S. Sidiroglou-Douskos, S. Misailovic, H. Hoffmann, and M. Rinard, "Managing performance vs. accuracy trade-offs with loop perforation," in *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering - SIGSOFT/FSE '11*, (Szeged, Hungary), p. 124, ACM Press, 2011.

[94] A. Lashgar, E. Atoofian, and A. Baniasadi, "Loop Perforation in OpenACC," in *2018 IEEE Intl Conf on Parallel & Distributed Processing with Applications, Ubiquitous Computing & Communications, Big Data & Cloud Computing, Social Computing & Networking, Sustainable Computing & Communications (ISPA/IUCC/BDCloud/SocialCom/SustainCom)*, (Melbourne, Australia), pp. 163–170, IEEE, Dec. 2018.

[95] S. Mittal, "A Survey of Techniques for Approximate Computing," *ACM Computing Surveys*, vol. 48, pp. 1–33, Mar. 2016.

[96] A. Yazdanbakhsh, G. Pekhimenko, B. Thwaites, H. Esmaeilzadeh, O. Mutlu, and T. C. Mowry, "RFVP: Rollback-Free Value Prediction with Safe-to-Approximate Loads," *ACM Trans. Archit. Code Optim.*, vol. 12, pp. 1–26, Jan. 2016.

[97] K. Georgios, C. Kokkala, and I. Stamoulis, "Clumsy Value Cache: An Approximate Memoization Technique for Mobile GPU Fragment Shaders," 2015.

[98] M. Franceschi, A. Nannarelli, and M. Valle, "Tunable Floating-Point for Artificial Neural Networks," in *2018 25th IEEE International Conference on Electronics, Circuits and Systems (ICECS)*, (Bordeaux), pp. 289–292, IEEE, Dec. 2018.

[99] T. M. Khoshgoftaar, J. Van Hulse, and A. Napolitano, "Supervised Neural Network Modeling: An Empirical Investigation Into Learning From Imbalanced Data With Labeling Errors," *IEEE Trans. Neural Netw.*, vol. 21, pp. 813–830, May 2010.

[100] G. Venkatesh, E. Nurvitadhi, and D. Marr, "Accelerating Deep Convolutional Networks using low-precision and sparsity," *arXiv:1610.00324 [cs]*, Oct. 2016. arXiv: 1610.00324.

[101] P. Judd, J. Albericio, T. Hetherington, T. Aamodt, N. E. Jerger, R. Urtasun, and A. Moshovos, "Reduced-Precision Strategies for Bounded Memory in Deep Neural Nets," *arXiv:1511.05236 [cs]*, Jan. 2016. arXiv: 1511.05236.

[102] H. Omar, M. Ahmad, and O. Khan, "GraphTuner: An Input Dependence Aware Loop Perforation Scheme for Efficient Execution of Approximated Graph Algorithms," in *2017 IEEE International Conference on Computer Design (ICCD)*, (Boston, MA), pp. 201–208, IEEE, Nov. 2017.

[103] J. S. Miguel, M. Badr, and N. E. Jerger, "Load Value Approximation," in *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*, (Cambridge, United Kingdom), pp. 127–139, IEEE, Dec. 2014.

[104] R. S. Amant, A. Yazdanbakhsh, J. Park, B. Thwaites, H. Esmaeilzadeh, A. Hassibi, L. Ceze, and D. Burger, "General-purpose code acceleration with limited-precision analog computation," in *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*, (Minneapolis, MN, USA), pp. 505–516, IEEE, June 2014.

[105] C. Alvarez, J. Corbal, and M. Valero, "Fuzzy Memoization for Floating-Point Multimedia Applications," *IEEE Trans. Comput.*, vol. 54, pp. 922–927, July 2005.

[106] Z. Xie, W. Dong, J. Liu, I. Peng, Y. Ma, and D. Li, "MD-HM: memoization-based molecular dynamics simulations on big memory system," in *Proceedings of the ACM International Conference on Supercomputing*, (Virtual Event USA), pp. 215–226, ACM, June 2021.

[107] A. Rahimi, L. Benini, and R. K. Gupta, "Spatial Memoization: Concurrent Instruction Reuse to Correct Timing Errors in SIMD Architectures," *IEEE Trans. Circuits Syst. II*, vol. 60, pp. 847–851, Dec. 2013.

[108] M. Samadi and S. Mahlke, "CPU-GPU collaboration for output quality monitoring," in *In Proceedings of the 1st Workshop on Approximate Computing across the System Stack. 1–3*, 2014.

[109] I. Goiri, R. Bianchini, S. Nagarakatte, and T. D. Nguyen, "ApproxHadoop: Bringing Approximations to MapReduce Frameworks," *SIGPLAN Not.*, vol. 50, pp. 383–397, May 2015.

[110] F. Sampaio, M. Shafique, B. Zatt, S. Bampi, and J. Henkel, "Approximation-aware Multi-Level Cells STT-RAM cache architecture," in *2015 International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES)*, (Amsterdam, Netherlands), pp. 79–88, IEEE, Oct. 2015.

[111] Q. Zhang, T. Wang, Y. Tian, F. Yuan, and Q. Xu, "ApproxANN: An Approximate Computing Framework for Artificial Neural Network," in *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2015*, (Grenoble, France), pp. 701–706, IEEE Conference Publications, 2015.

[112] W. Baek and T. M. Chilimbi, "Green: a framework for supporting energy-conscious programming using controlled approximation," in *Proceedings of the 2010 ACM SIGPLAN conference on Programming language design and implementation - PLDI '10*, (Toronto, Ontario, Canada), p. 198, ACM Press, 2010.

[113] Zidong Du, A. Lingamneni, Yunji Chen, K. Palem, O. Temam, and Chengyong Wu, "Leveraging the error resilience of machine-learning applications for designing highly energy efficient accelerators," in *2014 19th Asia and South Pacific Design Automation Conference (ASP-DAC)*, (Singapore), pp. 201–206, IEEE, Jan. 2014.

[114] Y. Cho and M. Lu, "A Reconfigurable Approximate Floating-Point Multiplier with kNN," in *2020 International SoC Design Conference (ISOCC)*, (Yeosu, Korea (South)), pp. 117–118, IEEE, Oct. 2020.

[115] T. Nomani, M. Mohsin, Z. Pervaiz, and M. Shafique, "xUAVs: Towards Efficient Approximate Computing for UAVs—Low Power Approximate Adders With Single LUT Delay for FPGA-Based Aerial Imaging Optimization," *IEEE Access*, vol. 8, pp. 102982–102996, 2020.

[116] D. Hernandez-Araya, J. Castro-Godinez, M. Shafique, and J. Henkel, "AUGER: A Tool for Generating Approximate Arithmetic Circuits," in *2020 IEEE 11th Latin American Symposium on Circuits & Systems (LASCAS)*, (San Jose, Costa Rica), pp. 1–4, IEEE, Feb. 2020.

[117] S.-H. Yang, C.-C. Chiu, C.-W. Chang, C.-M. Chen, C.-H. Meng, and K.-H. Chen, "87% Overall High Efficiency and 11 uA Ultra-Low Standby Current Derived by Overall Power Management in Laptops With Flexible Voltage Scaling and Dynamic Voltage Scaling Techniques," *IEEE Trans. Power Electron.*, vol. 31, pp. 3118–3127, Apr. 2016.

[118] A. Andrei, P. Eles, O. Jovanovic, M. Schmitz, J. Ogniewski, and Z. Peng, "Quasi-Static Voltage Scaling for Energy Minimization With Time Constraints," *IEEE Trans. VLSI Syst.*, vol. 19, pp. 10–23, Jan. 2011.

[119] L. Yao, D. I. Made, and Y. Gao, "A 83% peak efficiency 1.65 V to 11.4V dynamic voltage scaling supply for electrical stimulation applications in standard 0.18um CMOS process," in *2016 IEEE Asian Solid-State Circuits Conference (A-SSCC)*, (Toyama, Japan), pp. 205–208, IEEE, Nov. 2016.

[120] M. Ringenburg, A. Sampson, I. Ackerman, L. Ceze, and D. Grossman, "Monitoring and Debugging the Quality of Results in Approximate Programs," *SIGPLAN Not.*, vol. 50, pp. 399–411, May 2015.

[121] T. Moreau, M. Wyse, J. Nelson, A. Sampson, H. Esmaeilzadeh, L. Ceze, and M. Oskin, "SNNAP: Approximate computing on programmable SoCs via neural acceleration," in *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*, (Burlingame, CA, USA), pp. 603–614, IEEE, Feb. 2015.

[122] S. S. Sarwar, G. Srinivasan, B. Han, P. Wijesinghe, A. Jaiswal, P. Panda, A. Raghunathan, and K. Roy, "Energy Efficient Neural Computing: A Study of Cross-Layer Approximations," *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, vol. 8, pp. 796–809, Dec. 2018.

[123] J. Kung, D. Kim, and S. Mukhopadhyay, "A power-aware digital feedforward neural network platform with backpropagation driven approximate synapses," in *2015 IEEE/ACM International Symposium on Low Power Electronics and Design (ISLPED)*, (Rome, Italy), pp. 85–90, IEEE, July 2015.

[124] S. Venkataramani, A. Ranjan, K. Roy, and A. Raghunathan, "AxNN: energy-efficient neuromorphic systems using approximate computing," in *Proceedings of the 2014 international symposium on Low power electronics and design*, (La Jolla California USA), pp. 27–32, ACM, Aug. 2014.

[125] V. K. Chippa, D. Mohapatra, A. Raghunathan, K. Roy, and S. T. Chakradhar, "Scalable effort hardware design: exploiting algorithmic resilience for energy efficiency," in *Proceedings of the 47th Design Automation Conference on - DAC '10*, (Anaheim, California), p. 555, ACM Press, 2010.

[126] F. Yazici, A. S. YildiZ, A. Yazar, and E. G. Schmidt, "A Novel Scalable On-chip Switch Architecture with Quality of Service Support for Hardware Accelerated Cloud Data Centers," in *2020 IEEE 9th International Conference on Cloud Networking (CloudNet)*, (Piscataway, NJ, USA), pp. 1–4, IEEE, Nov. 2020.

[127] S. Eldridge, F. Raudies, D. Zou, and A. Joshi, "Neural network-based accelerators for transcendental function approximation," in *Proceedings of the 24th edition of the great lakes symposium on VLSI - GLSVLSI '14*, (Houston, Texas, USA), pp. 169–174, ACM Press, 2014.

[128] V. Wong and M. Horowitz, "Soft Error Resilience of Probabilistic Inference Applications," in *IN PROCEEDINGS OF THE WORKSHOP ON SYSTEM EFFECTS OF LOGIC SOFT ERRORS*, 2006.

[129] L. Leem, Hyungmin Cho, J. Bau, Q. A. Jacobson, and S. Mitra, "ERSA: Error Resilient System Architecture for probabilistic applications," in *2010 Design, Automation & Test in Europe Conference & Exhibition (DATE 2010)*, (Dresden), pp. 1560–1565, IEEE, Mar. 2010.

[130] S. Misailovic, S. Sidiroglou, H. Hoffmann, and M. Rinard, "Quality of service profiling," in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - ICSE '10*, vol. 1, (Cape Town, South Africa), p. 25, ACM Press, 2010.

[131] P. Roy, R. Ray, C. Wang, and W. F. Wong, "ASAC: automatic sensitivity analysis for approximate computing," *ACM SIGPLAN Notices*, vol. 49, pp. 95–104, May 2014.

[132] J. Ludwig, S. Nawab, and A. Chandrakasan, "Low-power digital filtering using approximate processing," *IEEE Journal of Solid-State Circuits*, vol. 31, pp. 395–400, Mar. 1996.

[133] E. Nogues, D. Menard, and M. Pelcat, "Algorithmic-Level Approximate Computing Applied to Energy Efficient Hevc Decoding," *IEEE Transactions on Emerging Topics in Computing*, pp. 1–1, 2016.

[134] M. Osta, A. Ibrahim, H. Chible, and M. Valle, "Approximate Multipliers Based on Inexact Adders for Energy Efficient Data Processing," in *2017 New Generation of CAS (NGCAS)*, (Genova, Italy), pp. 125–128, IEEE, Sept. 2017.

[135] F. Muslim, A. Demian, L. Ma, L. Lavagno, and A. Qamar, "Energy-efficient FPGA Implementation of the k-Nearest Neighbors Algorithm Using OpenCL," pp. 141–145, Oct. 2016.

[136] S. Hahn, J. Reineke, and R. Wilhelm, "Towards compositionality in execution time analysis: definition and challenges," *ACM SIGBED Review*, vol. 12, pp. 28–36, Mar. 2015.

[137] M. Rizk, A. Baghdadi, M. Jézéquel, Y. Mohanna, and Y. Atat, "Efficient quantization and fixed-point representation for MIMO turbo-detection and turbo-demapping," *EURASIP Journal on Embedded Systems*, vol. 2017, Dec. 2017.

[138] C.-C. Chang and C.-J. Lin, "LIBSVM: A library for support vector machines," *ACM Transactions on Intelligent Systems and Technology*, vol. 2, pp. 1–27, Apr. 2011.

[139] Z. Liu, B. Wu, W. Luo, X. Yang, W. Liu, and K.-T. Cheng, "Bi-Real Net: Enhancing the Performance of 1-bit CNNs With Improved Representational Capability and Advanced Training Algorithm," *arXiv:1808.00278 [cs]*, Sept. 2018.

[140] X. Xilinx, *Vivado Design Suite User guide, High-Level Synthesis.*

[141] H. Fares, L. Seminara, A. Ibrahim, M. Franceschi, L. Pinna, M. Valle, S. Dosen, and D. Farina, "Distributed Sensing and Stimulation Systems for Sense of Touch Restoration in Prosthetics," in *2017 New Generation of CAS (NGCAS)*, (Genova, Italy), pp. 177–180, IEEE, Sept. 2017.

[142] J. Sun, W. Du, and N. Shi, "A Survey of kNN Algorithm," *Inf Eng Appl Comput*, vol. 1, May 2018.

[143] Zhe-Hao Li, Ji-Fang Jin, Xue-Gong Zhou, and Zhi-Hua Feng, "K-nearest neighbor algorithm implementation on FPGA using high level synthesis," in *2016 13th IEEE International Conference on Solid-State and Integrated Circuit Technology (ICSICT)*, (Hangzhou, China), pp. 600–602, IEEE, Oct. 2016.

[144] J. Saikia, S. Yin, Z. Jiang, M. Seok, and J.-s. Seo, "K-Nearest Neighbor Hardware Accelerator Using In-Memory Computing SRAM," in *2019 IEEE/ACM International Symposium on Low Power Electronics and Design (ISLPED)*, (Lausanne, Switzerland), pp. 1–6, IEEE, July 2019.

[145] D. Jamma, O. Ahmed, S. Areibi, G. Grewal, and N. Molloy, "Design exploration of ASIP architectures for the K-Nearest Neighbor machine-learning algorithm," in *2016 28th International Conference on Microelectronics (ICM)*, (Giza, Egypt), pp. 57–60, IEEE, Dec. 2016.

[146] D. Jamma, O. Ahmed, S. Areibi, and G. Grewal, "Hardware accelerators for the K-nearest neighbor algorithm using high level synthesis," in *2017 29th International Conference on Microelectronics (ICM)*, (Beirut, Lebanon), pp. 1–4, IEEE, Dec. 2017.

[147] F. B. Muslim, L. Ma, M. Roozmeh, and L. Lavagno, "Efficient FPGA Implementation of OpenCL High-Performance Computing Applications via High-Level Synthesis," *IEEE Access*, vol. 5, pp. 2747–2762, 2017.

[148] Q. Mu, L. Cui, and Y. Song, "The implementation and optimization of Bitonic sort algorithm based on CUDA," *arXiv:1506.01446 [cs]*, June 2015.

[149] H. Younes, A. Ibrahim, M. Rizk, and M. Valle, "Algorithmic Level Approximate Computing for Machine Learning Classifiers," in *2019 26th IEEE International Conference on Electronics, Circuits and Systems (ICECS)*, (Genoa, Italy), pp. 113–114, IEEE, Nov. 2019.

[150] N. Jamali and C. Sammut, "Majority Voting: Material Classification by Tactile Sensing Using Surface Texture," *IEEE Transactions on Robotics*, vol. 27, pp. 508–521, June 2011.

[151] J. Kwiatkowski, D. Cockburn, and V. Duchaine, "Grasp stability assessment through the fusion of proprioception and tactile signals using convolutional neural networks," in *2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, (Vancouver, BC), pp. 286–292, IEEE, Sept. 2017.

[152] H. Younes, A. Ibrahim, M. Rizk, and M. Valle, "Algorithmic-Level Approximate Tensorial SVM Using High-Level Synthesis on FPGA," *Electronics*, vol. 10, p. 205, Jan. 2021.

[153] *Weka 3 - Data Mining with Open Source Machine Learning Software in Java*.

[154] *TE0726 Resources - Public Docs - Trenz Electronic Wiki*.

[155] NVIDIA, *NVIDIA System Management Interface*. June 2012.

[156] L. Sánchez, J. Ranilla, and A. Cocaña-Fernández, "Eecluster: An energyefficient tool for managing hpc clusters," *Annals of Multicore and GPU Programming*, vol. 2, no. 1, pp. 15–24, 2015.

[157] S. Cook, "Memory Handling with CUDA," in *CUDA Programming*, pp. 107–202, Elsevier, 2013.

[158] M. A. Mohsin and D. G. Perera, "An FPGA-Based Hardware Accelerator for K-Nearest Neighbor Classification for Machine Learning on Mobile Devices," in *Proceedings of the 9th International Symposium on Highly-Efficient Accelerators and Reconfigurable Technologies*, (Toronto ON Canada), pp. 1–7, ACM, June 2018.

[159] D. Tao, X. Li, X. Wu, W. Hu, and S. J. Maybank, "Supervised tensor learning," *Knowledge and Information Systems*, vol. 13, pp. 1–42, Sept. 2007.

[160] B. Zhou, R. Brent, and M. Kahn, "Efficient one-sided Jacobi algorithms for singular value decomposition and the symmetric eigenproblem," in *Proceedings 1st International Conference on Algorithms and Architectures for Parallel Processing*, vol. 1, (Brisbane, Qld., Australia), pp. 256–262, IEEE, 1995.

[161] B. Yang and J. F. Böhme, "Reducing the Computations of the Singular Value Decomposition Array Given by Brent and Luk," *SIAM J. Matrix Anal. & Appl.*, vol. 12, pp. 713–725, Oct. 1991.

[162] R. P. Brent and F. T. Luk, "The Solution of Singular-Value and Symmetric Eigenvalue Problems on Multiprocessor Arrays," *SIAM J. Sci. and Stat. Comput.*, vol. 6, pp. 69–84, Jan. 1985.

[163] J. R. Cavallaro and F. T. Luk, "Architectures For A Cordic SVD Processor," (San Diego), p. 45, Mar. 1986.

[164] J.-M. Delosme, "CORDIC Algorithms: Theory And Extensions," (San Diego), p. 131, Nov. 1989.

[165] D. Milford and M. Sandell, "Singular value decomposition using an array of CORDIC processors," *Signal Processing*, vol. 102, pp. 163–170, Sept. 2014.

[166] A. Ibrahim, M. Valle, L. Noli, and H. Chible, "Singular value decomposition FPGA implementation for tactile data processing," in *2015 IEEE 13th International New Circuits and Systems Conference (NEWCAS)*, (Grenoble, France), pp. 1–4, IEEE, June 2015.

[167] S. Zhang, X. Tian, C. Xiong, J. Tian, and D. Ming, "Fast Implementation for the Singular Value and Eigenvalue Decomposition Based on FPGA," *Chinese Journal of Electronics*, vol. 26, pp. 132–136, Jan. 2017.

[168] T. Jiang, F. Xie, S. Yuan, S. Yao, K. Wu, and X. Wang, "Implementation of Matrix SVD Decomposition Module for Subspace Channel Estimation," in *2019 IEEE 3rd Information Technology, Networking, Electronic and Automation Control Conference (ITNEC)*, (Chengdu, China), pp. 1096–1102, IEEE, Mar. 2019.

[169] C. Deng, M. Yin, X.-Y. Liu, X. Wang, and B. Yuan, "High-performance Hardware Architecture for Tensor Singular Value Decomposition: Invited Paper," in *2019 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, (Westminster, CO, USA), pp. 1–6, IEEE, Nov. 2019.

[170] A. Ibrahim, M. Valle, L. Noli, and H. Chible, "FPGA implementation of fixed point CORDIC-SVD for E-skin systems," in *2015 11th Conference on Ph.D. Research in Microelectronics and Electronics (PRIME)*, (Glasgow, United Kingdom), pp. 318–321, IEEE, June 2015.

[171] N. Samardzija and R. L. Waterland, "A neural network for computing eigenvectors and eigenvalues," *Biol. Cybern.*, vol. 65, pp. 211–214, Aug. 1991.

[172] Z. Yi, Y. Fu, and H. J. Tang, "Neural networks based approach for computing eigenvectors and eigenvalues of symmetric matrix," *Computers & Mathematics with Applications*, vol. 47, pp. 1155–1164, Apr. 2004.

[173] Y. Tang and J. Li, "Another neural network based approach for computing eigenvalues and eigenvectors of real skew-symmetric matrices," *Computers & Mathematics with Applications*, vol. 60, pp. 1385–1392, Sept. 2010.

[174] J. Qiu, H. Wang, J. Lu, B. Zhang, and K.-L. Du, "Neural Network Implementations for PCA and ItsExtensions," *ISRN Artificial Intelligence*, vol. 2012, pp. 1–19, 2012.

[175] H. Bui and S. Tahar, "Design and synthesis of an IEEE-754 exponential function," in *Engineering Solutions for the Next Millennium. 1999 IEEE Canadian Conference on Electrical and Computer Engineering (Cat. No.99TH8411)*, vol. 1, (Edmonton, Alta., Canada), pp. 450–455, IEEE, 1999.

[176] M. Osta, M. Alameh, H. Younes, A. Ibrahim, and M. Valle, "Energy Efficient Implementation of Machine Learning Algorithms on Hardware Platforms," (Genova, Italy), Nov. 2019.

[177] I. L. Jernelv, D. R. Hjelme, Y. Matsuura, and A. Aksnes, "Convolutional neural networks for classification and regression analysis of one-dimensional spectral data," *arXiv:2005.07530 [physics, stat]*, May 2020.

[178] M. Fernández-Delgado, M. Sirsat, E. Cernadas, S. Alawadi, S. Barro, and M. Febrero-Bande, "An extensive experimental survey of regression methods," *Neural Networks*, vol. 111, pp. 11–34, Mar. 2019.

[179] A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, and H. Adam, "MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications," *arXiv:1704.04861 [cs]*, Apr. 2017.

[180] X. Zhang, X. Zhou, M. Lin, and J. Sun, "ShuffleNet: An Extremely Efficient Convolutional Neural Network for Mobile Devices," *arXiv:1707.01083 [cs]*, Dec. 2017.

[181] I. Freeman, L. Roese-Koerner, and A. Kummert, "EffNet: An Efficient Structure for Convolutional Neural Networks," *arXiv:1801.06434 [cs]*, June 2018.

[182] L. Yang and A. Shami, "On hyperparameter optimization of machine learning algorithms: Theory and practice," *Neurocomputing*, vol. 415, pp. 295–316, Nov. 2020.

[183] K. He, X. Zhang, S. Ren, and J. Sun, "Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification," *arXiv:1502.01852 [cs]*, Feb. 2015.

[184] H. Younes, A. Ibrahim, M. Rizk, and M. Valle, "Data Oriented Approximate K-Nearest Neighbor Classifier for Touch Modality Recognition," in *2019 15th Conference on Ph.D Research in Microelectronics and Electronics (PRIME)*, (Lausanne, Switzerland), pp. 241–244, IEEE, July 2019.

[185] M. Alameh, Y. Abbass, A. Ibrahim, G. Moser, and M. Valle, "Touch Modality Classification Using Recurrent Neural Networks," *IEEE Sensors J.*, vol. 21, pp. 9983–9993, Apr. 2021.

[186] C. Alippi, S. Disabato, and M. Roveri, "Moving Convolutional Neural Networks to Embedded Systems: The AlexNet and VGG-16 Case," in *2018 17th ACM/IEEE International Conference on Information Processing in Sensor Networks (IPSN)*, (Porto), pp. 212–223, IEEE, Apr. 2018.

[187] Y. He, X. Zhang, and J. Sun, "Channel Pruning for Accelerating Very Deep Neural Networks," in *2017 IEEE International Conference on Computer Vision (ICCV)*, (Venice), pp. 1398–1406, IEEE, Oct. 2017.

[188] J. Yim, D. Joo, J. Bae, and J. Kim, "A Gift from Knowledge Distillation: Fast Optimization, Network Minimization and Transfer Learning," in *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, (Honolulu, HI), pp. 7130–7138, IEEE, July 2017.

[189] Y. Gong, L. Liu, M. Yang, and L. Bourdev, "Compressing Deep Convolutional Networks using Vector Quantization," *arXiv:1412.6115 [cs]*, Dec. 2014.

[190] Y. Bengio, N. Léonard, and A. Courville, "Estimating or Propagating Gradients Through Stochastic Neurons for Conditional Computation," *arXiv:1308.3432 [cs]*, Aug. 2013.

[191] D. P. Kingma and J. Ba, "Adam: A Method for Stochastic Optimization," *arXiv:1412.6980 [cs]*, Jan. 2017.

[192] H. Yonekawa and H. Nakahara, "On-Chip Memory Based Binarized Convolutional Deep Neural Network Applying Batch Normalization Free Technique on an FPGA," in *2017 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, (Orlando / Buena Vista, FL, USA), pp. 98–105, IEEE, May 2017.

[193] U. S. Patankar, M. E. Flores, and A. Koel, "Division algorithms - From Past to Present Chance to Improve Area Time and Complexity for Digital Applications," in *2020 IEEE Latin America Electron Devices Conference (LAEDC)*, (San Jose, Costa Rica), pp. 1–4, IEEE, Feb. 2020.

[194] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, "Attention Is All You Need," *arXiv:1706.03762 [cs]*, Dec. 2017. arXiv: 1706.03762.

[195] A. Dosovitskiy, L. Beyer, A. Kolesnikov, D. Weissenborn, X. Zhai, T. Unterthiner, M. Dehghani, M. Minderer, G. Heigold, S. Gelly, J. Uszkoreit, and N. Houlsby, "An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale," *arXiv:2010.11929 [cs]*, June 2021. arXiv: 2010.11929.

[196] P. Huang, C. Wang, W. Liu, F. Qiao, and F. Lombardi, "A Hardware/Software Co-Design Methodology for Adaptive Approximate Computing in clustering and ANN Learning," *IEEE Open J. Comput. Soc.*, vol. 2, pp. 38–52, 2021.

Appendix A

Hardware Accelerator Design using Vivado Suite

A.1 Hardware Design using Register-Transfer Level (RTL)

1. Create a new Vivado Project and choose your board (e.g. Zynqberry).
2. Create a new block design.
3. Add the Zynq processing system then:
   a. Connect the FCLK_CLK0 to M_AXI__GP0_ACLK.
   b. Run Connection Automation.
4. Right click on the design in the source pane and select “create HDL wrapper”.
5. Choose Tools » Create and Package New IP » Create AXI4 Interface.
6. A new Vivado IP project is open, inside it: Choose add source » create a new one: VHDL code shown in Figure A.1. -
   In the source pane, expand the item: my_multiplier_v1_0_S00_AXI_inst and add the following changes:
   - Find the line with the “begin” keyword and add the code shown in Figure A.2 just above it to declare the multiplier and the output signal.
   - Find the line that says “– Add user logic here” and add the code shown in Figure A.3 below it to instantiate the multiplier.
   - Find this line of code “reg_data_out <= slv_reg1;” and replace it with “reg_data_out <= multiplier_out;”.

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

entity multiplier is
  port(
    clk : in std_logic;
    a   : in std_logic_vector(15 downto 0);
    b   : in std_logic_vector(15 downto 0);
    p   : out std_logic_vector(31 downto 0)
  );
end multiplier;

architecture IMP of multiplier is

begin
  process (clk)
  begin
    if clk'event and clk = '1' then
      p <= a * b;
    end if;
  end process;
end IMP;
```

Fig. A.1 VHDL Code of a multiplier

```
signal multiplier_out : std_logic_vector(31 downto 0);
component multiplier
port (
  clk: in std_logic;
  a: in std_logic_VECTOR(15 downto 0);
  b: in std_logic_VECTOR(15 downto 0);
  p: out std_logic_VECTOR(31 downto 0));
end component;
```

Fig. A.2 Multiplier Setup

```
multiplier_0: multiplier
port map (
  clk => S_AXI_ACLK,
  a => slv_reg0(31 downto 16),
  b => slv_reg0(15 downto 0),
  p => multiplier_out);
```

Fig. A.3 Port Mapping of the Multiplier Entity

- In the process statement just a few lines above, replace “slv_reg1” with “multiplier_out”. You should notice that the “multiplier.vhd” file has been integrated into the hierarchy.
- In the package pane:
Click on “IP File Group” and select “Merge changes from IP File. . . .”
Click on Review and Package. Repackage your IP, save and close.

7. In the main Vivado project, add the new created IP to your design and run Connection Automation. The complete design is shown in Figure A.4.

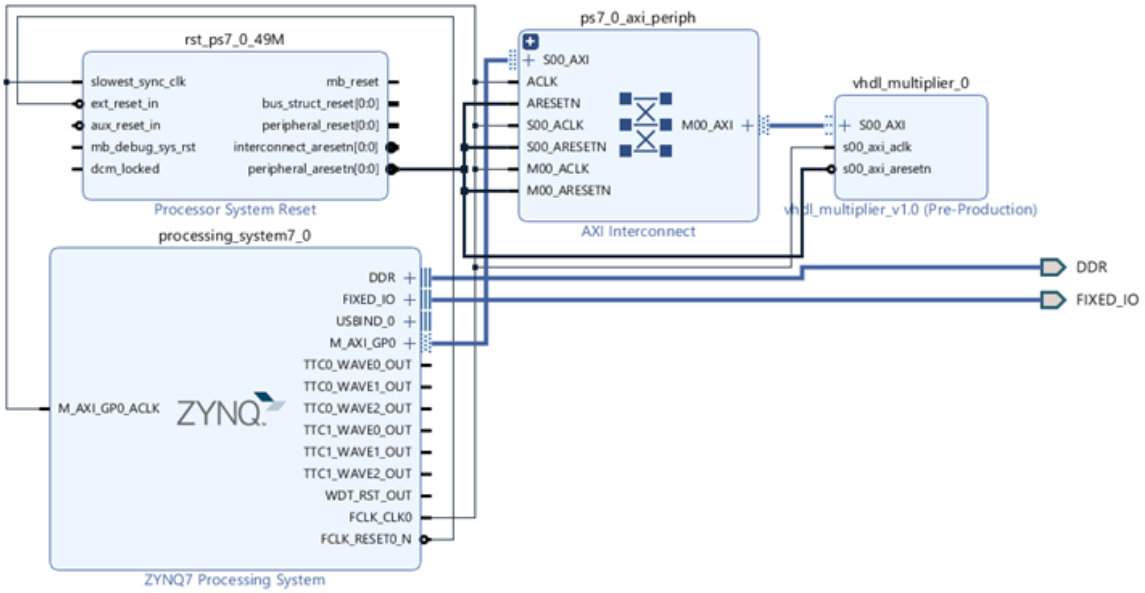


Fig. A.4 Complete RTL Design Block Diagram

8. From “Program and Debug”, select Generate bits.
9. Select File » Export Hardware and check “Include bitstream”.
10. Select File » Launch SDK.
11. In the SDK Application:
a. Select File » New Application Project.
b.Keep everything as default (except C/C++ based on your reference), and Click Next.
c. Choose the hello World Template.
d. Paste the code shown in Figure A.5 inside Helloworld.c: This code saves two numbers into a 32-bit register (num1 in LSB, num2 in MSB) then perform the multiplication using the VHDL IP.

```c
#include "platform.h"
#include "xbasic_types.h"
#include "xparameters.h"

Xuint32 *baseaddr_p = (Xuint32 *) XPAR_VHDL_MULTIPLIER_0_S00_AXI_BASEADDR;

int main()
{
init_platform();

xil_printf("Multiplier Test\n\r");

// Write multiplier inputs to register 0
*(baseaddr_p+0) = 0x00020003;
xil_printf("Wrote: 0x%08x \n\r", *(baseaddr_p+0));

// Read multiplier output from register 1
xil_printf("Read : 0x%08x \n\r", *(baseaddr_p+1));

xil_printf("End of test\n\n\r");

return 0;
}
```

Fig. A.5 Multiplier Code in Vivado SDK using RTL

12. Program the FPGA
13. Select Run » Run As » Launch on Hardware GDB
14. Open serial connection using the SDK terminal on the PORT number. See the results.

A.2 Software Design using High Level Synthesis (HLS)

1. Create new project in Vivado HLS.
2. Create new C file in the source tab that includes a simple multiplication function.
3. Use: #pragma HLS INTERFACE s_axilite port=a bundle="CTRLS" directive to export the block as AXI interface.
4. Create new C file in the testbench tab that includes the main() function.
5. Run C simulation, RTL/C simulation and finally Export RTL IP.
6. In Vivado:
   a. Repeat steps 1 to 4 from section A.1
   b. Select Tools » Setting » IP » Repository and add the path to the HLS exported block.
   c. Run Connection Automation. The complete design is shown in Figure A.6.
7. Repeat steps 8-to-11(c) from section A.1.

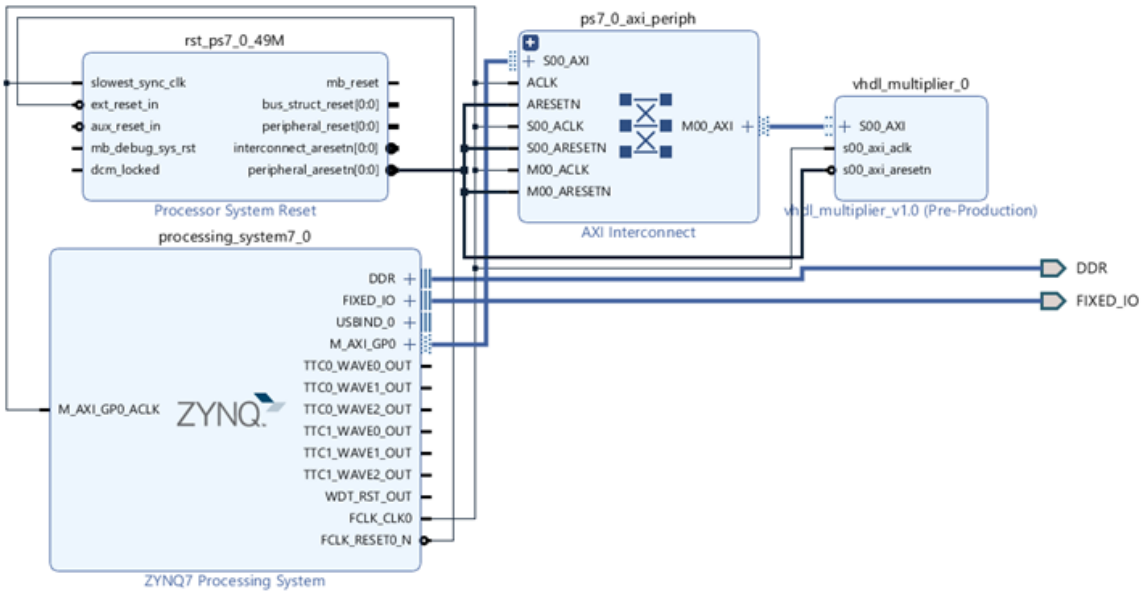


Fig. A.6 Complete HLS Design Block Diagram

8. In SDK, paste the code shown in Figure A.7 in the “helloworld.c”.

```c
#include "platform.h"
#include "xbasic_types.h"
#include "xparameters.h" // Contains definitions for all peripherals
#include "xhls_multiplier.h" // Contains hls_multiplier

// This is how to call any HLS block
XHls_multiplier do_hls_multiplier;
XHls_multiplier_Config *do_hls_multiplier_cfg;

// this function initializes the HLS block
void init_HLS_multiplier(){
        int status;
        // Create hls_multiplier pointer
        do_hls_multiplier_cfg = XHls_multiplier_LookupConfig(
        XPAR_HLS_MULTIPLIER_0_DEVICE_ID);

        if (!do_hls_multiplier_cfg) {
            xil_printf(
                    "Error loading configuration for do_hls_multiplier_cfg \n\r");
        }
        status = XHls_multiplier_CfgInitialize(&do_hls_multiplier,
                do_hls_multiplier_cfg);
        if (status != XST_SUCCESS) {
            xil_printf("Error initializing for do_hls_multiplier \n\r");
        }
        XHls_multiplier_Initialize(&do_hls_multiplier,
        XPAR_HLS_MULTIPLIER_0_DEVICE_ID); // this is optional in this case
}
```

```c
int main()
{
    init_platform();
    init_HLS_multiplier();

    unsigned int p;
    p = 0;
    while(1){
        int a = 2, b=3;
    // Write multiplier inputs to register 0
    XHls_multiplier_Set_a(&do_hls_multiplier, a);
    XHls_multiplier_Set_b(&do_hls_multiplier, b);
    xil_printf("Write a: 0x%08x \n\r", a);
    xil_printf("Write b: 0x%08x \n\r", b);

    // Start hls_multiplier
    XHls_multiplier_Start(&do_hls_multiplier);
    xil_printf("Started hls_multiplier \n\r");

    // Wait until it's done (optional here)
    while (!XHls_multiplier_IsDone(&do_hls_multiplier));

    // Get hls_multiplier returned value
    p = XHls_multiplier_Get_return(&do_hls_multiplier);

    xil_printf("Read p: 0x%08x \n\r", p);

    xil_printf("End of test HLS_MULTIPLIER \n\n\r");
    }
    cleanup_platform();
    return 0;
}
```

Fig. A.7 Multiplier Code in Vivado SDK using HLS

9. Repeat steps 12-to-14 from section A.1.

A.3 Software/hardware Co-Design using HLS and RTL

1. Follow the same steps as previous sections to obtain the accelerator design shown in Figure A.8.

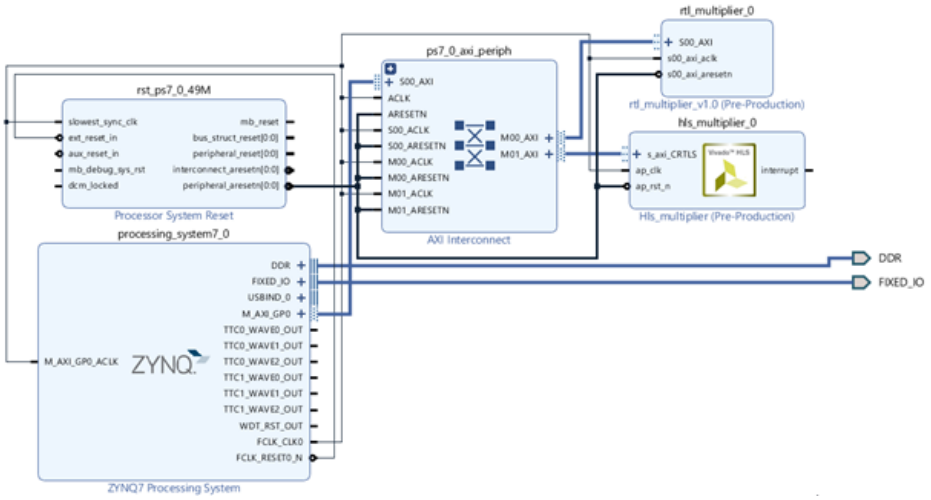


Fig. A.8 Software/Hardware Multiplier Accelerator

2. In SDK, paste the code shown in Figures A.9, A.10, A.11, A.12, and A.13 in the "helloworld.c".

```
#include "platform.h"
#include "xbasic_types.h"
#include "xparameters.h" // Contains definitions for all peripherals
#include "xhls_multiplier.h" // Contains hls_multiplier macros and functions

// we will use the Base Address of the RTL_MULTIPLIER
Xuint32 *baseaddr_rtl_multiplier =
        (Xuint32 *) XPAR_RTL_MULTIPLIER_0_S00_AXI_BASEADDR;

// global for HLS MULTIPLIER
XHls_multiplier do_hls_multiplier;
XHls_multiplier_Config *do_hls_multiplier_cfg;

// function that prompts user for 2 numbers (ints)
void get_inputs(int *c, int*d) {
    int a, b;
    // get first operand
    xil_printf("Enter operand A: ");
    scanf("%d", &a);
    xil_printf("%d\n\r", a);
    // get second operand
    xil_printf("Enter operand B: ");
    scanf("%d", &b);
    xil_printf("%d\n\r", b);
    // return the two numbers
    *c = a;
    *d = b;
    return;
}
```

Fig. A.9 Multiplier Code in Vivado SDK using HLS and RTL (1)

```
void multiply_rtl(int a, int b) {
    // concatenate the two integers to a 32-bit value to be passed to RTL multiplier
    int passing_int = (a << 16) + b;
    // Write multiplier inputs to register 0

    *(baseaddr_rtl_multiplier + 0) = passing_int;
    xil_printf("Wrote to register 0: 0x%08x \n\r",
           *(baseaddr_rtl_multiplier + 0));

    // Read multiplier output from register 1
    xil_printf("Read from register 1: 0x%08x \n\r",
           *(baseaddr_rtl_multiplier + 1));

    xil_printf("End of test RTL_MULTIPLIER \n\n\r");

}
```

Fig. A.10 Multiplier Code in Vivado SDK using HLS and RTL (2)

```
void init_HLS_multiplier(){
        // Vivado HLS generates
        int status;
        // Create hls_multiplier pointer
        do_hls_multiplier_cfg = XHls_multiplier_LookupConfig(
        XPAR_HLS_MULTIPLIER_0_DEVICE_ID);

        if (!do_hls_multiplier_cfg) {
            xil_printf(
                    "Error loading configuration for do_hls_multiplier_cfg \n\r");
        }

        status = XHls_multiplier_CfgInitialize(&do_hls_multiplier,
                do_hls_multiplier_cfg);
        if (status != XST_SUCCESS) {
            xil_printf("Error initializing for do_hls_multiplier \n\r");
        }

        XHls_multiplier_Initialize(&do_hls_multiplier,
        XPAR_HLS_MULTIPLIER_0_DEVICE_ID); // this is optional in this case
}
```

Fig. A.11 Multiplier Code in Vivado SDK using HLS and RTL (3)

```
// function that multiplies with HLS multiplier
void multiply_hls(int a, int b) {
    unsigned int p;
    p = 0;

    // Write multiplier inputs to register 0
    XHls_multiplier_Set_a(&do_hls_multiplier, a);
    XHls_multiplier_Set_b(&do_hls_multiplier, b);
    xil_printf("Write a: 0x%08x \n\r", a);
    xil_printf("Write b: 0x%08x \n\r", b);

    // Start hls_multiplier
    XHls_multiplier_Start(&do_hls_multiplier);
    xil_printf("Started hls_multiplier \n\r");

    // Wait until it's done (optional here)
    while (!XHls_multiplier_IsDone(&do_hls_multiplier))
        ;

    // Get hls_multiplier returned value
    p = XHls_multiplier_Get_return(&do_hls_multiplier);

    xil_printf("Read p: 0x%08x \n\r", p);

    xil_printf("End of test HLS_MULTIPLIER \n\n\r");
}
```

Fig. A.12 Multiplier Code in Vivado SDK using HLS and RTL (4)

3. Repeat steps 12-to-14 from section A.1.

```
int main() {
    // setup
    init_platform();
    init_HLS_multiplier();
    int *c, *d;
    // Infinite loop steps: (1) test RTL multiplier with user inputs and (2) test HLS multiplier with user inputs
    while (1) {
        ////////////////////////////////////////////////////////////////////////////////
        // RTL MULTIPLIER TEST
        xil_printf("Performing a test of the RTL_MULTIPLIER... \n\r");
        // prompt user for 2 numbers (to be used for RTL multiplication)
        get_inputs(c, d);
        // perform RTL multiplication
        multiply_rtl(*c, *d);

        ////////////////////////////////////////////////////////////////////////////////
        // HLS MULTIPLIER TEST
        xil_printf("Performing a test of the HLS_MULTIPLIER... \n\r");
        // prompt user for 2 numbers (to be used for HLS multiplication)
        get_inputs(c, d);
        // perform HLS multiplication
        multiply_hls(*c, *d);
    }

    cleanup_platform();
    return 0;
}
```

Fig. A.13 Multiplier Code in Vivado SDK using HLS and RTL (5)

This process can be performed with verilog and Vivado Vitis with the same procedure.