# Massively-Parallel and Concurrent SVM Architectures

P.B.A. Phear, B.E. (Hons.)

Thesis submitted to the University of Nottingham
for the degree of Master of Philosophy

March 2018

# Abstract

This work presents several Support Vector Machine (SVM) architectures developed by the Author with the intent of exploiting the inherent parallel structures and potential-concurrency underpinning the SVM's mathematical operation. Two SVM training sub-system prototypes are presented - a brute-force search classification training architecture, and, Artificial Neural Network (ANN)-mapped optimisation architectures for both SVM classification training and SVM regression training. This work also proposes and prototypes a set of parallelised SVM Digital Signal Processor (DSP) pipeline architectures. The parallelised SVM DSP pipeline architectures have been modelled in C and implemented in VHDL for the synthesis and fitting on an Altera Stratix V FPGA. Each system presented in this work has been applied to a problem domain application appropriate to the SVM system's architectural limitations - including the novel application of the SVM as a chaotic and non-linear system parameter-identification tool.

The SVM brute-force search classification training architecture has been modelled for datasets of 2 dimensions and composed of linear and non-linear problems requiring only 4 support vectors by utilising the linear kernel and the polynomial kernel respectively. The system has been implemented in Matlab and non-exhaustively verified using the holdout method with a trivial linearly separable classification problem dataset and a trivial non-linear XOR classification problem dataset. While the architecture was a feasible design for software-based implementations targeting 2-dimensional datasets the architectural complexity and unmanageable number of parallelisable operations introduced by increasing data-dimensionality and the number of support vectors subsequently resulted in the Author pursuing different parallelised-architecture strategies.

Two distinct ANN-mapped optimisation strategies developed and proposed for SVM classification training and SVM regression training have been modelled in Matlab; the architectures have been designed such that any dimensionality dataset can be applied by configuring the appropriate dimensionality and support vector parameters. Through Monte-Carlo testing using the datasets examined in this work the gain parameters inherent in the architectural design of the systems were found to be difficult to tune, and, system convergence to acceptable sets of training support vectors were unachieved. The ANN-mapped optimisation strategies were thus deemed inappropriate for SVM training with the applied datasets without more design effort and architectural modification work.

The parallelised SVM DSP pipeline architecture prototypes data-set dimensionality, support vector set counts, and latency ranges follow. In each case the Field Programmable Gate Array (FPGA) pipeline prototype latency unsurprisingly outclassed the corresponding C-software model execution times by at least 3 orders of magnitude. The SVM classification training DSP pipeline FPGA prototypes are compatible with data-sets spanning 2 to 8 dimensions, support vector sets of up to 16 support vectors, and have a pipeline latency range spanning from a minimum of 0.18 microseconds to a maximum of 0.28 microseconds. The SVM classification function evaluation DSP pipeline FPGA prototypes are compatible with data-sets spanning 2 to 8 dimensions, support vector sets of up to 32 support vectors, and have a pipeline latency range spanning from a minimum of 0.16 microseconds to a maximum of 0.24 microseconds. The SVM regression training DSP pipeline FPGA prototypes are compatible with data-sets spanning 2 to 8 dimensions, support vector sets of up to 16 support vectors, and have a pipeline latency range spanning from a minimum of 0.20 microseconds to a maximum of 0.30 microseconds. The SVM regression function evaluation DSP pipeline FPGA prototypes are compatible with data-sets spanning 2 to 8 dimensions, support vector sets of up to 16 support vectors, and have a pipeline latency range spanning from a minimum of 0.20 microseconds to a maximum of 0.30 microseconds.

Finally, utilising LIBSVM training and the parallelised SVM DSP pipeline function evaluation architecture prototypes, SVM classification and SVM regression was successfully applied to Rajkumar's oil and gas pipeline fault detection and failure system legacy data-set yielding excellent results. Also utilising LIBSVM training, and, the parallelised SVM DSP pipeline function evaluation architecture prototypes, both SVM classification and SVM regression was applied to several chaotic systems as a feasibility study into the application of the SVM machine learning paradigm for chaotic and non-linear dynamical system parameter-identification. SVM classification was applied to the Lorenz Attractor and an ANN-based chaotic oscillator to a reasonably acceptable degree of success. SVM classification was applied to the Mackey-Glass attractor yielding poor results. SVM regression was applied Lorenz Attractor and an ANN-based chaotic oscillator yielding average but encouraging results. SVM regression was applied to the Mackey-Glass attractor yielding poor results.

# Contents

# Preface

## 0.1 Supporting Publications

- R. K. Rajkumar, P. B. A. Phear, D. Isa, W. Y. Wan, and N. A. Akram, "Real-time pipeline monitoring system and method thereof," Malaysian Patent Application PI 2015704444, December 4, 2015.
- P. B. A. Phear, R. K. Rajkumar, and D. Isa, "Efficient non-iterative fixed-period SVM training architecture for FPGAs," in *Proc. of the 39th Annu. Conf. of the IEEE Industrial Electronics Society (IECON 2013)*, Vienna, Austria, November 2013.

## 0.2 List of Figures

## 0.3 List of Tables

## 0.4 List of Algorithm and Code Listings

# Glossary

## 0.5 Notation

| | |
|---|---|
| $\mathbb{R}^n$ | Euclidean $n$ space |
| $C_k$ | Scalar constant $C_k$ |
| $x_n$ | Scalar value $x_n$ |
| $\|x_n\|$ | The absolute value of scalar $x_n$ |
| $\bar{\boldsymbol{x}}_k$ | Column vector $\bar{\boldsymbol{x}}_k$ in $\mathbb{R}^n$ |
| $\boldsymbol{x}$ | A set, or ensemble, of $k$ column vectors $\bar{\boldsymbol{x}}_0$, $\bar{\boldsymbol{x}}_1$, $\cdots$ , $\bar{\boldsymbol{x}}_k$ in $\mathbb{R}^n$ |
| $\|\bar{\boldsymbol{x}}_k\|$ | The cardinality of vector $\bar{\boldsymbol{x}}_k$ |
| $\bar{\boldsymbol{x}}_k^T$ | The transpose of vector $\bar{\boldsymbol{x}}_k$ |
| $\|\bar{\boldsymbol{x}}_k\|$ | The norm of vector $\bar{\boldsymbol{x}}_k$ |
| $\bar{\boldsymbol{x}}_j \bullet \bar{\boldsymbol{x}}_k$ | The dot-product of vectors $\bar{\boldsymbol{x}}_j$ and $\bar{\boldsymbol{x}}_k$ |
| $\boldsymbol{I}$ | The identity matrix $\boldsymbol{I}$ |
| $\boldsymbol{A}$ | Matrix $\boldsymbol{A}$ |
| $\boldsymbol{A}^{-1}$ | The inverse of square matrix $\boldsymbol{A}$ |
| $rank(\boldsymbol{A})$ | The rank of matrix $\boldsymbol{A}$ |
| $det(\boldsymbol{A})$ | The determinant of square matrix $\boldsymbol{A}$ |
| $\|\boldsymbol{A}\|$ | The determinant of square matrix $\boldsymbol{A}$ |
| $\boldsymbol{J}$ | Jacobian matrix of partial derivatives |
| $\lambda_{\boldsymbol{A}}$ | Eigenvalue $\lambda$ of square matrix $\boldsymbol{A}$ |
| $\bar{\boldsymbol{e}}_\lambda$ | Eigenvector $\bar{\boldsymbol{e}}_\lambda$ corresponding to eigenvalue $\lambda$ |
| $\lambda$ | Eigenvalue, Lyapunov exponent, or Lagrange multiplier |
| $\alpha$ | Lagrange multiplier |

| | |
|---|---|
| $\|D\|$ | The cardinality of set $D$, the number of elements in set $D$ |
| $\circledast$ | The convolution operator |
| $\mathcal{Z}\{\cdot\}$ | The z-transform operator |
| $\mathcal{Z}\{\cdot\}^{-1}$ | The inverse z-transform operator |
| $\mathcal{F}\{\cdot\}$ | The Discrete Fourier transform operator |
| $\mathcal{F}\{\cdot\}^{-1}$ | The Discrete inverse Fourier transform operator |
| $\mathcal{Q}\{\cdot\}$ | The quantisation operator |
| $\mathcal{C}\{S\}$ | the convex hull of subset $S$ |

## 0.6 Acronyms and Abbreviations

| | |
|---|---|
| AI | Artificial Intelligence |
| ANN | Artificial Neural Network |
| ANND | Artificial Neural Network Chaotic Oscillator Data-Set |
| ASIC | Application-Specific Integrated-Circuit |
| BJT | Bipolar Junction Transistor |
| CMOS | Complimentary Metal-Oxide Semiconductor |
| CPLD | Complex Programmable Logic Device |
| CPU | Central Processing Unit |
| DFT | Discrete Fourier Transform |
| DRAM | Dynamic-RAM |
| DSP | Digital Signal Processor |
| DUT | Device Under Test |
| DVCS | Distributed Version Control System |
| EEPROM | Electronically-Erasable Programmable Read-Only Memory |
| EM | Expectation Maximisation |
| EPROM | Erasable Programmable Read-Only Memory |
| ERM | Empirical Risk Minimisation |
| FPGA | Field Programmable Gate Array |

| | |
|---|---|
| FSM | Finite State Machine |
| GAL | Generic Array Logic |
| GPGPU | General-purpose Computing on Graphics Processing Units |
| GRG | Generalised Reduced Gradient |
| HDL | Hardware Description Language |
| HLS | High-Level Synthesis |
| HPC | High-performance Computing / Supercomputer |
| IC | Integrated Circuit |
| i.i.d. | independent and identically distributed |
| IP | Intellectual Property |
| KKT | Karush-Kuhn-Tucker |
| kPCA | Kernel Principal Component Analysis |
| LAD | Lorenz Attractor Data-Set |
| LPD | Legacy Oil and Gas Pipeline Data-Set |
| LUT | Look Up Table |
| MAC | Multiply-Accumulate |
| MGAD | Mackey-Glass Attractor Data-Set |
| MLP | Multilayer Perceptron |
| NAN | Not-A-Number |
| NPA | Nearest Point Algorithm |
| PAL | Programmable Array Logic |
| PCA | Principal Component Analysis |
| PLA | Programmable Logic Array |
| PROM | Programmable Read-Only Memory |
| PSD | Power Spectral Density |
| QP | Quadratic Programming |
| RAM | Random Access Memory |
| ROM | Read-Only Memory |

| | |
|---|---|
| RTL | Register Transfer Logic |
| TTL | Transistor-Transistor Logic |
| RNN | Recurrent Neural Network |
| SMO | Sequential Minimal Optimisation |
| SoC | System on a Chip |
| SOM | Self Organising Map |
| SOP | Sum-Of-Products |
| SPLD | Simple Programmable Logic Device |
| SQNR | Signal-to-Quantisation Noise Ratio |
| SRAM | Static-RAM |
| SRM | Structural Risk Minimisation |
| SVM | Support Vector Machine |
| VHDL | Very-high-speed integrated circuit HDL |

# Chapter 1

# Introduction

The Support Vector Machine (SVM) is a powerful and well understood supervised learning tool in the field of Machine Learning and applied Artificial Intelligence (AI). The SVM has been used to address a diverse range of real-world problems and applications including image and object recognition, speech recognition, e-mail filtering, DNA sequencing, and skin-cancer melanoma detection [1], [2], [3]. The SVM has its developmental roots in Vapnik's work on Statistical Learning Theory [4] and shares many of its architectural linear-mathematical underpinnings as the Artificial Neural Network (ANN) machine learning paradigm first proposed by Rosenblatt [5]. This thesis presents SVM training and function evaluation architectures developed with the intent of exploiting the inherent parallel structures and potential-concurrency underpinning the SVM's mathematical basis and thus suitability for implementation as massively parallelised FPGA hardware. Each system presented in this work has been applied to a problem domain application appropriate to the SVM system's architectural limitations - including the novel application of the SVM as a chaotic and non-linear system parameter-identification tool.

Coupled with the advent of the transistor, the closing decades of the Twentieth Century saw the genesis of the long-time-considered science-fiction-fantasy known as AI; now a vibrant, practical, and application-rich discipline. However, compared to the capacity of biological based intellect, Man's silicone-borne AI proves a very primitive species indeed. Nevertheless, through Man's voracious desire to assimilate and apply wisdom, the advantage endowed to biological intelligence through the untimely chaotic process of Darwinian Evolution is decaying at an ever increasing rate.

The eminent rule-of-thumb known as Moore's Law states: the transistor count on an integrated circuit will double roughly every two years [6]. Moore's law has held-true with respect to developments centred in Man's reality for well over forty years. Couple this fact with Man's industrious application of acquired wisdom, extrapolate into the not-to-distant future, and not unlike the evolutionary leaps forward observed in biological systems, a radical electronics-based AI transformation of science-fiction-like proportions, as illustrated in Fig. 1.1, can be predicted with frighteningly high confidence [7], [8], [9].

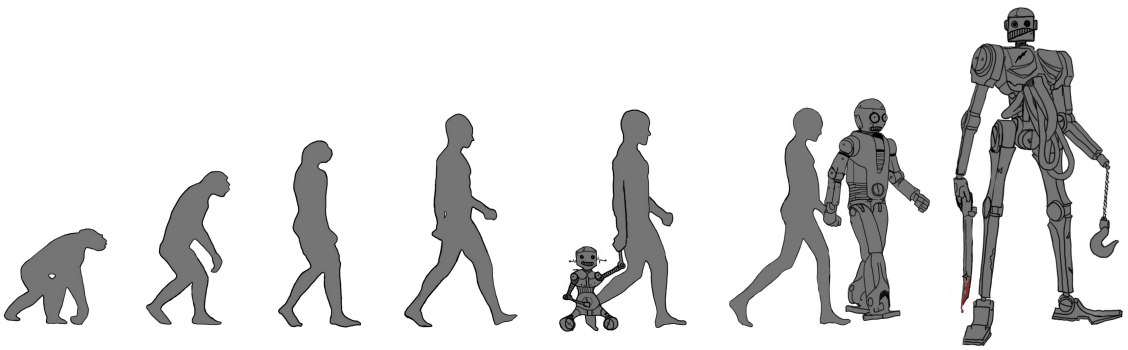


***Figure 1.1:*** *Mankind's past, present, and possible future, illustrated as discrete evolutionary leaps forward, from left to right, in time.*

Amongst the varyingly successful AI or Machine Learning strategies developed and studied by Man, one intermittently popular and reoccurring paradigm, first proposed by Rosenblatt in 1957, models neural processes commonly found in the brain-matter of living biological beings. Such a network of neurons has become known as and generally referred to as an Artificial Neural Network (ANN). Rosenblatt dubbed his ANN system The Perceptron [5]. Figure 1.2 illustrates both a biological neuron structure and the Perceptron's neuron structure. ANNs exploit the use of multidimensional mathematics known as linear algebra; neuron inputs and synaptic connection weights are represented as rational numbers and neuron firing-function evaluation is essentially a trivial numerical computation easily implemented in standard signal processing technologies.

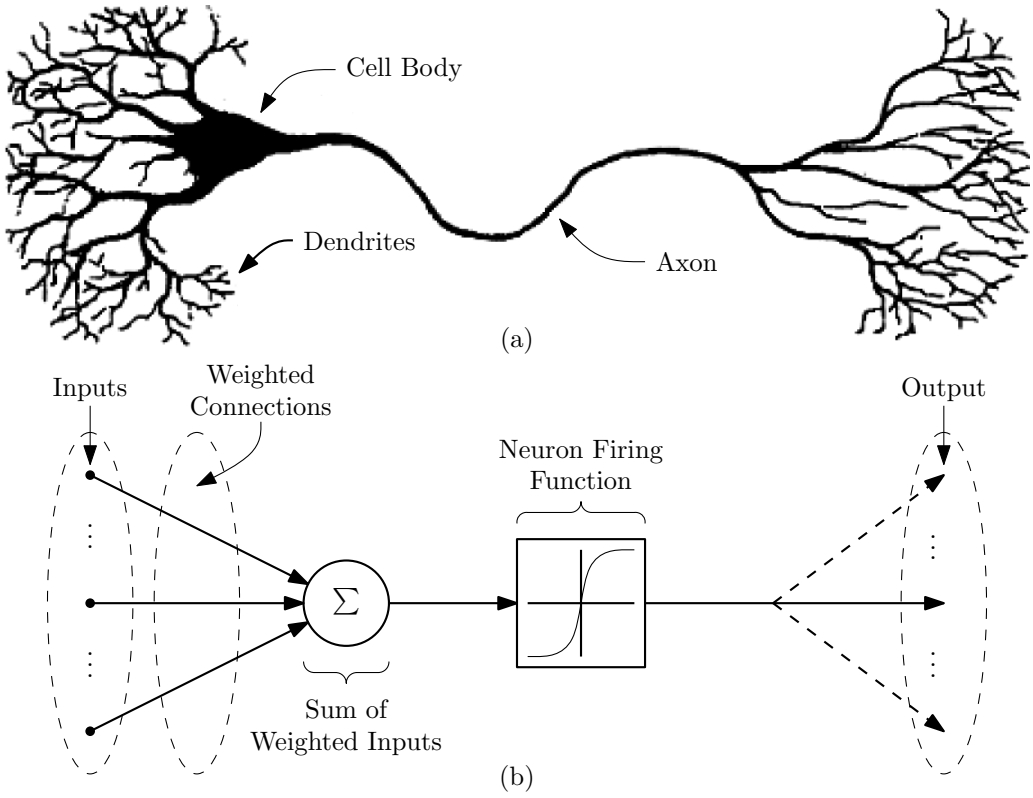



***Figure 1.2:*** *The Perceptron was modelled on biological neuron function; (a) a biological neuron structure and (b) Rosenblatt's artificial neuron structure.*

A year later in 1958 another new synthetic organism was conceived [10]. Man would come to call this new beast the Integrated Circuit (IC). An IC comprises the interconnection of entirely silicon-composed resistor, capacitor, and semiconductor components to form fully-functional circuits within a minute silicon die [11]. The birth of the IC beaconed and elevated transistor technology to almost in-expendable prominence, spawned another electronics-miniaturisation revolution, enabled even smaller and more efficient computing devices, and made feasible and helped-facilitate Man's first expedition to the moon. The IC would come to be regarded as a major achievement of 20th Century Electrical and Electronic Engineering [12].

Meanwhile, some began to notice that Rosenblatt's Perceptron had weaknesses. Borrowing from the biologically-inspired adage *"monkey see, monkey do,"* ANN systems require explicit training to perform their proposed task. By applying training sets as inputs and evaluating the outputs, then modifying connection weights accordingly, ANNs can be trained to perform almost any classification task. This form of training is known as *Supervised Training* [13]. Published in 1969, Minsky and Papert's book entitled *Perceptrons: An Introduction to Computational Geometry* detailed the Perceptron's shortcomings [14]. The most significant of these shortcomings was the Perceptron's inability to classify linearly inseparable data, even with appropriate training. This handicap, also known as the *Exclusive-OR problem*, arises when two independent classes of data-points cannot be separated by a simple straight line. Subsequently ANN research was temporarily abandoned resulting in an era colloquially known as the first "AI winter" [15].

Elsewhere during the early 1960s, a young mathematician-come-meteorologist at the Massachusetts Institute of Technology named Edward Norton Lorenz began playing with a simplified nonlinear atmospheric weather pattern model on his Royal McBee LGP-30 vacuum-tube computer. Figure 1.3 shows a Royal McBee LGP-30 vacuum-tube computer complete with human operator; the cumbersome, unwieldy, and undependable state-of-the-art of the times. Lorenz would program his model to simulate different weather patterns by entering varying parameters and initial conditions and observing the evolution of a system via the print-out user-interface.

***Figure 1.3:*** *Royal McBee LGP-30 vacuum-tube computer, as used by Edward Lorenz, complete with operator.*

Before long Lorenz wanted to repeat a certain weather pattern he had observed before, but this time over an extended period. To save some time he decided to start the simulation halfway through the initial run, using the original run's half-way point values from the print-out record as initial conditions. The new simulation initially tracked the original run, as expected, but then unexpectedly began to diverge before displaying completely different response. The error between simulation-runs initially led Lorenz to believe his computer was faulty; a vacuum-tube must have blown - not an unlikely situation to find oneself in when using an ever-unreliable LGP-30 vacuum-tube computer. On further inspection and thought Lorenz realised this was not the case; the error was his. The program would perform numerical calculations in six significant-figures, however, to save space, only three significant-figures were ever printed. Lorenz had entered the rounded-to-three-significant-figure initial conditions with the incorrect expectation that such an error would be insignificant [16], [17]. Lorenz had discovered his system of nonlinear differential equations, his bounded deterministic simplified atmospheric weather model, displayed a sensitive dependence to initial conditions [18]. Lorenz would later call this sensitive dependence to initial conditions *The Butterfly Effect*; a butterfly flaps its wings today, causing a tornado tomorrow.

Lorenz had inadvertently pioneered a new scientific field: *Chaos Theory.* Lorenz's accidental discovery of deterministic turbulent complexity within a nonlinear dynamic system, later becoming known as the Lorenz Attractor shown in Fig. 1.4, would lay hidden in a niche meteorology journal for years to come. Academic credit from a wider scientific audience, for a time, would remain unfulfilled. The complex systems displayed in nature were historically considered by the classical Newtonian-Laplacian school of thought, otherwise known as the greater physics community, as nature's disorder, stochastic, and a nightmarish monstrosity [17]. For years to come this position would remain so, informing a general disdain towards Chaotic Systems research and those that dared to believe and pursue such research directions. However this predictable (as seen time and time again, ad nauseam, throughout the ages) conservative close-minded scientific attitude and in-

discriminate dismissal would not last forever. Other reputable and established scientists from many varied and traditionally unacquainted scientific arena began to notice similar patterns; reoccurring order within apparent disorder and simple nonlinear system dynamics displaying complex and erratic responses from only slight variations in initial conditions.

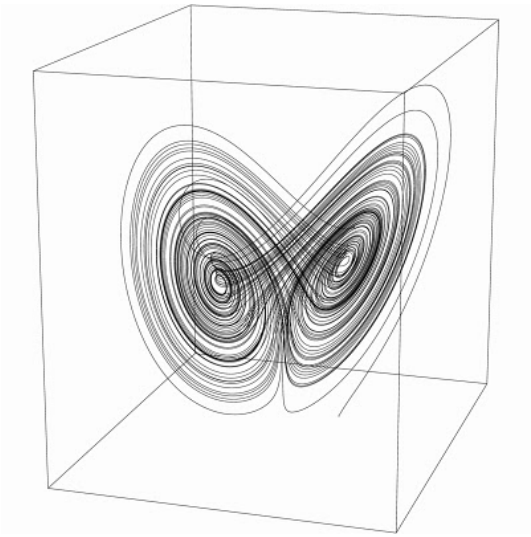


***Figure 1.4:*** *A three-dimensional state-space or phase-space reconstruction of the Lorenz Attractor.*

The AI winter finally passed in the early to mid 1980s on the back of further consistent evolutionary leaps forward in IC technology. Integrated digital-logic had transformed computing into an in-expensive, attainable, and legitimate pursuit for anyone outside of a research laboratory. The personal-computing revolution had begun. With the number of personal computers available on the consumer market and in private homes increasing, the number and computational power of these devices inside of the research laboratory increased even more so. With this increase of computational abilities, the burden of traditionally computationally-heavy research was relieved, and subjects that were once regarded as unwieldy, thus justifiably abandoned or avoided, were resurrected and revitalised with renewed gusto. Both Chaotic and Nonlinear Systems and ANN research were such subject areas that saw a significant increase in interest, research attention, and an evolution into mature and legitimate fields of study in their own right.

Benoît B. Mandelbrot, a French-American mathematician working for IBM and on secondment to Harvard University, was one of many utilising computers to visualise simple iterated processes and mathematical mappings. In the mid 1970s Mandelbrot coined the term *Fractal* to describe reoccurring self-similar structures observed at different scales within a pattern, a reoccurring theme throughout his research. Using the computing power available to him, Mandelbrot began experimenting with a class of fractal shapes known as a *Julia Set*, first discovered and rendered meticulously by hand during World War I by the French mathematicians Gaston Julia and Pierre Fatou [17]. Mandelbrot's ensuing variation on this theme was the iteration of $z \to z^2 + c$ of every complex num-

ber $c$ on the complex plane. The generated fractal pattern became universally known as the *Mandelbrot Set*, shown in Fig. 1.5; it would come to serve as a complete catalogue of all the Julia Set fractals [19]. The image would also signify the reluctant acceptance of Chaotic and Nonlinear Systems theory amongst the greater scientific community, and serve as the poster child of mathematics, science, and engineering for years to come [16].

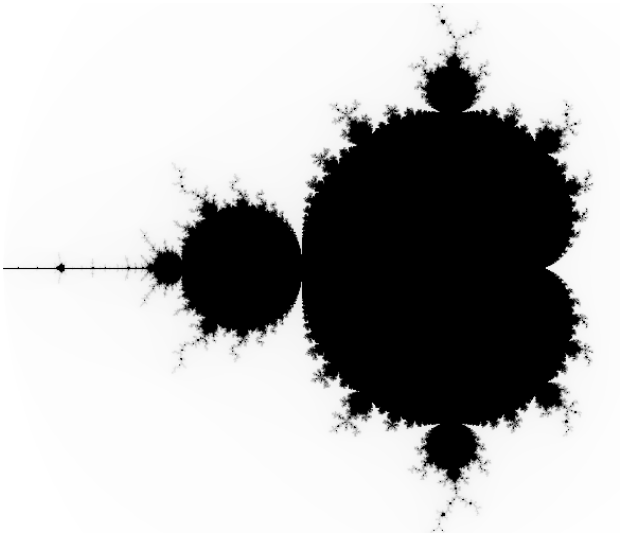


***Figure 1.5:*** *The Mandelbrot Set fractal, the iteration of $z \to z^2 + c$ on every complex number $c$ on the complex plane; $c$ belongs to the set, and hence is coloured black, if the iterated result $z$ remains bounded, oscillates chaotically, or does not tend to infinity.*

Also during the 1980s researchers began to employ the term *Machine Learning* to avoid any negative connotations AI had earned during the field's adolescence. In due course the Perceptron's Exclusive-OR problem was solved and its bad publicity more-or-less absolved. Unsupervised learning was achieved through the Generalized Hebbian Algorithm based upon only inputs and outputs of a single layer ANN [20] [21]. Also by adding more layers of neurons to the Perceptron, a Multilayer Perceptron (MLP) as shown in Fig. 1.6, many of the flaws presented by Minsky and Papert's famous treatise were overcome or disproved [22], [23]. The MLP also saw the development of the now-indispensable supervised learning technique named *Back-propagation for* ANN *training.*

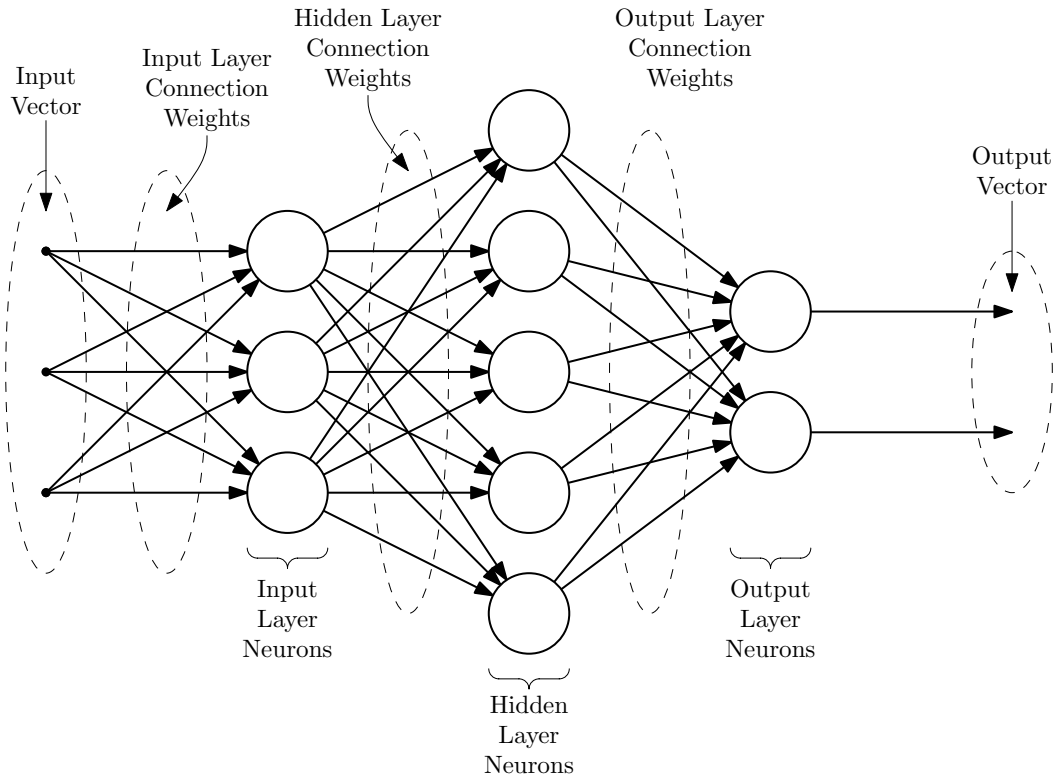



***Figure 1.6:*** *An example of a Multilayer Perceptron.*

It was during this time of renewed-research themes and computing developments that IC-based digital logic was to undergo an orthogonal evolutionary branching toward a new paradigm. As digital circuits grew larger, the wiring and interconnecting of individual discrete components became unwieldy, and an unpleasant and undesirable pursuit. Creating and manufacturing an Application-Specific Integrated-Circuit (ASIC) was enormously expensive and only feasible for those already entrenched in the industry. Digital logic designers dreamt of computer based circuit design. Hardware engineers were jealous of the software engineers' workflow, design, and development paradigm. They yearned for painless circuit instantiation, testing, and on-the-fly bug-fixing. The industry was calling for new technology in the form of an infallible, scalable, and reprogrammable hardware platform of mammoth proportions. Ross Freeman and Bernard Vonderschmitt founded Xilinx Inc. on these principles, and in 1985 invented the first commercially viable FPGA logic device [24]. The FPGA would continue the electronic industry's habit of revolution and innovation; a tradition that still persists today with each successive generation of FPGA technology.

As time passed and the 1980s drew to a close, many more ANN system topologies were developed, and the underlying mathematical operation of these networks were expanded upon. Thus new systems capable of increasingly complex tasks emerged from the resurrected field of research. These tasks included function approximation, regression analysis, pattern and sequence recognition, digital signal processing, system control and supervision, and even time-series prediction. Also the theoretical groundwork for another new

Machine Learning paradigm was undergoing gestation. Vladimir N. Vapnik was a Soviet-born mathematician working at the Adaptive Systems Research Department at AT&T Bell Labs in Holmdel, New Jersey, USA. Here he developed the underlying statistics-based theory of a new Machine Learning paradigm, the Support Vector Machine (SVM) [4], detailed in diverse rigour, in a variety of publications [25], [26], [27], [28].

SVMs and ANNs both employ very similar mathematics, however, operate on differing principles. Because of their mathematical similarity and common conceptual heritage SVMs are often regarded a class of ANN. Hence SVM research findings are generally encountered in ANN periodicals. Traditionally both ANN and SVM systems have been implemented in software and used to solve problems across many fields. Not as frequently these learning machines have been implemented on an FPGA platform. By bringing together the fields of Signal Processing, Digital FPGA Hardware Design, Machine Learning, and Chaotic Systems, and as an analogue to the chaotic nature to the evolution of biological intelligence, this research aims to further advance Man's own synthetic learning machines.

## 1.1 Problem Statement

A software-based SVM model has been developed in the Department of Electrical and Electronic Engineering at the University of Nottingham by Rajkumar [29]. The SVM model can classify oil and gas pipeline defects, determine the location of any defect along the length of the pipe and about the pipe's circumference, and, predict approximate time to pipeline failure due to unrestrained corrosion. Rajkumar's SVM model achieves pseudo-unsupervised learning by utilising $k$-means clustering to supply appropriate labels to the SVM's training subsystem. However Rajkumar's model suffers a handicap that prevents its use in a practical context - it lacks an implementation that operates in real-time, driven by real-time process-instrumentation, and thus suitable for real-world process applications.

## 1.2 Research Objectives

The primary objective of this research project was to investigate, implement, and apply SVM classification and regression machine learning paradigms utilising massively-parallel and concurrent architectures, improving Rajkumar's SVM model's performance, and thus approach true real-time operation. This objective was met by exploring parallelised SVM architectures and implementing the SVM classification and SVM regression subsystems' underlying mathematics as massively parallelised FPGA-based DSP pipeline hardware. Additionally novel SVM classification and regression training strategies composed of parallel architectural-structures and functional-blocks - thus suitable for implementation as parallelised hardware - were also investigated, developed, and modelled.

## 1.3 System Overview

Figure 1.7 provides a high-level block-diagram overview of the FPGA-based SVM hardware and accompanying tools, models, and software that has been developed, implemented, and employed as part of this work. The systems illustrated in Fig. 1.7 include the SVM DSP pipelines and auxiliary subsystems, system memory management and control software, data-set management and result analysis tools, and, SVM training and DSP pipeline model software.

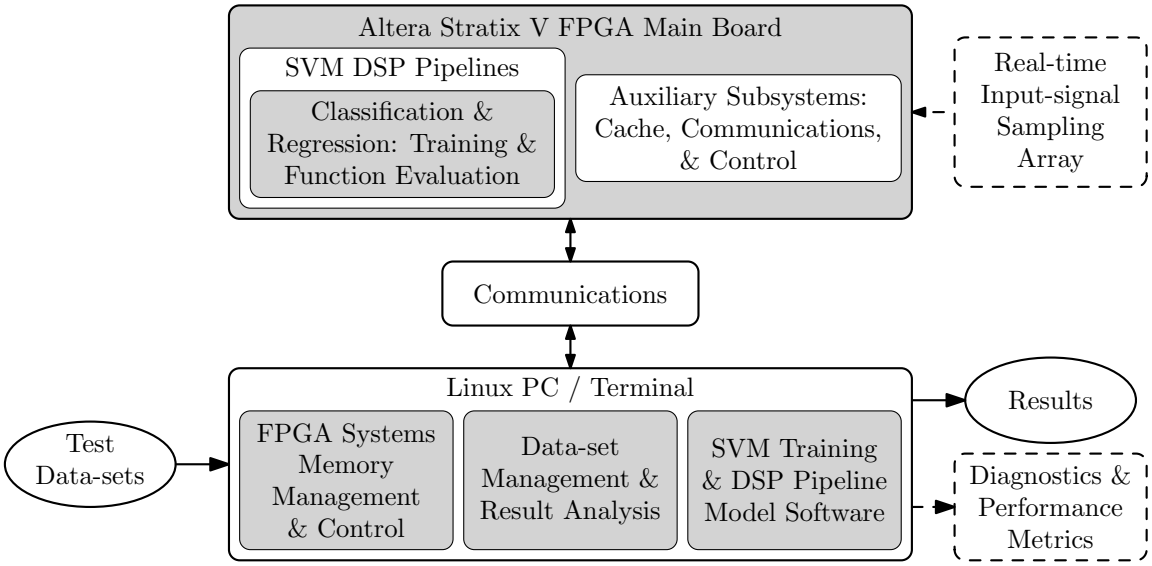



***Figure 1.7:*** *High-level block-diagram overview of the SVM Hardware Architecture and Auxiliary subsystems and accompanying tools, models, and software.*

## 1.4 Research Scope

Support Vector Machine Classification and Support Vector Machine Regression has been investigated, implemented, and applied as part of the body of this research. The developed SVM training and function evaluation models and systems were applied to the following application domains.

As an extension of Dr. Rajprasad Kumar Rajkumar's prior research, the SVM models and pipeline subsystems have been applied to the oil and gas pipeline fault detection and failure prediction system data-set. Also, feasibility studies and testing into the application of the SVM machine learning paradigm in nonlinear and chaotic dynamical systems domain was conducted. SVM Classification and Regression of three chaotic systems - the Lorenz Attractor, the Mackey-Glass attractor, and an ANN-based chaotic oscillator, have been conducted.

As a study of the SVM's performance as a function of input data dimension, and, as a verification and validation of each developed SVM pipeline-architecture's generalised-design reconfigurability, each application domain test-ensemble examines variations in

data dimensionality through various techniques and mechanisms specific to its host domain.

In all application domains the SVM system's primary performance metric is that of real-time operation; all other metrics of interest, both technically quantitative and qualitative, have been measured based principally upon the system's real-time performance and secondarily informed by the specific application domain and its typical engineering constraints.

## 1.5 Subject Area Contributions

The scope of this work encompasses four subject areas; FPGA Hardware Design and Implementation, Digital Signal Processing, Machine Learning, and Chaotic and Nonlinear Systems. The Venn diagram in Fig. 1.8 illustrates this relationship, culminating with this work's tangible output, the SVM hardware architectures. This work incorporates elements of, and in-turn contributes back to, these research subject areas.

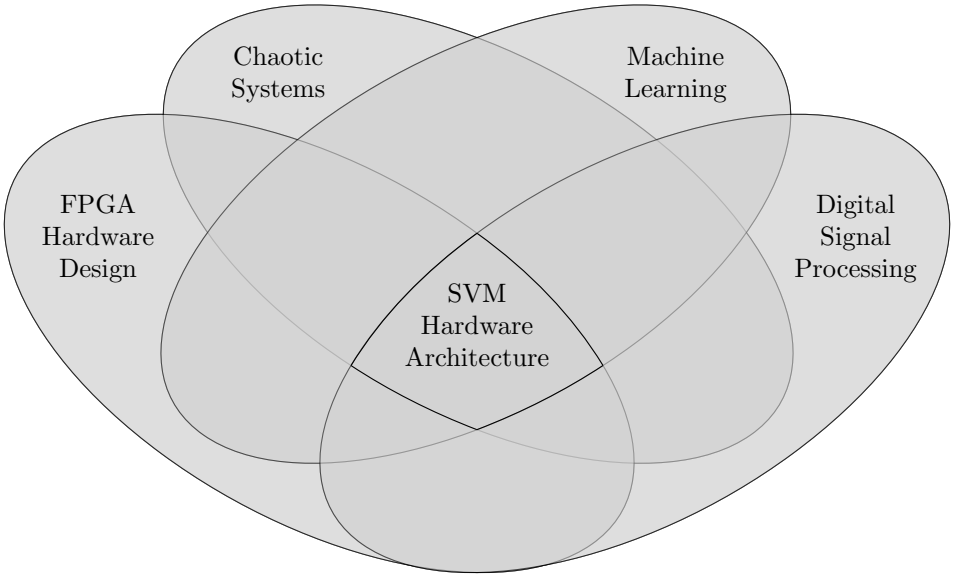



***Figure 1.8:*** *Venn diagram illustrating the four subject areas covered in this work's scope and its subsequent research contributions.*

## 1.6 Organisation

This thesis is organised into the following chapters; Preface, Glossary, Chapter 1: Introduction, Chapter 2: Preliminaries, Chapter 3: Literature Review, Chapter 4: System Architecture and Scientific Method, Chapter 5: Results, Chapter 6: Discussion, and Chapter 7: Conclusion. Finally Appendices followed by References are found at the end of this thesis.

A Table of Contents can be found at the beginning of this thesis and Appendices and References can found at the end of this thesis.

The Preface contains a list of supporting publications derived from this work and lists of all figures and tables found within this thesis. The Glossary contains a list of mathematical and scientific notation, and, a list of acronyms and abbreviations used throughout this thesis.

Chapter 1 introduces the scope and goals of this research project.

Chapter 2 defines the fundamental concepts used and applied freely and frequently throughout the thesis. These concepts are included in this chapter to alleviate clutter and thus afford coherent development of new concepts, and, to serve as a convenient reference for the reader.

Chapter 3 contains the most recent review of currently available literature pertaining to the scope of currently completed work. It identifies gaps in the literature that have been addressed, and, presents concepts imparted through the literature that have been applied and extended throughout this body of work.

Chapter 4 describes the design and development of systems implemented throughout this work, and, provides an overview of data-sets and the scientific method applied to test these system implementations.

Chapter 5 presents technical specifications and measured data pertaining to the systems designed and developed as part of this work, and, the results to the application of the data-sets and scientific method described in the previous chapter.

Chapter 6 provides an extensive discussion of the system designs and implementations presented in Chapter 4 and a discussion of the technical specifications, measured data, and results presented in the Chapter 5.

Chapter 7 briefly summarises the research project objectives, goals, and outcomes and thus concludes the thesis.

The Appendices contain a listing of all DSP instructions used in the developed SVM pipelines and the VHDL entities of the prototyped SVM DSP pipelines.

Finally the References chapter contains a list, in the standard IEEE citation style, of all publications and literature explicitly cited in this thesis.

# Chapter 2

# Preliminaries

This chapter defines the mathematical conventions, principles, and methods used throughout the development of the SVM systems presented in this thesis. These conventions are provided as a concise and succinct reference to mathematical operations and methods, including nomenclature, utilised and developed in later chapters; both proofs and derivations have been omitted.

Support Vector Machine operation is composed of the multidimensional mathematics of Linear Algebra. Section 2.1 provides an overview of this field of mathematics. Vector and matrix conventions, vector space definitions, lines, planes, and hyperplanes theory, concepts and nomenclature used throughout this thesis are covered.

SVM training is an optimisation problem. Thus optimisation problem theory and accompanying nomenclature are defined in Section 2.2.

Finally Section 2.3 Taylor Series and Section 2.4 are presented as they provide key functional definitions and approximations useful for the implementation of SVM in digital systems.

## 2.1 Linear Algebra

### 2.1.1 Vectors and Matrices

A *vector* is a one dimensional array of $n$ elements, such as a data time series or a position vector in $\mathbb{R}^n$, and will be represented by a bar-accented bold-type lower-case Roman or Greek character. E.g. vector $\bar{\boldsymbol{x}}$ is a one dimensional column vector of $n$ elements $\{x_1, x_2, \cdots, x_n\}$ in $\mathbb{R}^n$ and is defined as

$$\bar{\boldsymbol{x}} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}, \tag{2.1}$$

The cardinality of $\bar{\boldsymbol{x}}$, written $|\bar{\boldsymbol{x}}|$, is the number of elements in $\bar{\boldsymbol{x}}$, and is therefore $n$. The *transpose* of $\bar{\boldsymbol{x}}$, a one dimensional row vector of $n$ elements in $\mathbb{R}^n$, represented as $\bar{\boldsymbol{x}}^T$, is defined as

$$\bar{\boldsymbol{x}}^T = \begin{bmatrix} x_1 & x_2 & \cdots & x_n \end{bmatrix}. \tag{2.2}$$

Given vector $\bar{\boldsymbol{x}}$ defines a position vector in $\mathbb{R}^n$, the length of $\bar{\boldsymbol{x}}$, known as the *norm* and represented by $\|\bar{\boldsymbol{x}}\|$, is defined as

$$\|\bar{\boldsymbol{x}}\| = \sqrt{\bar{\boldsymbol{x}} \bullet \bar{\boldsymbol{x}}}. \tag{2.3}$$

A *matrix* is a two dimensional array of $m \times n$ components, such as multi-sensor data time series or an ANN connection weight matrix, and is represented by a bold-type upper-case Roman or Greek character. E.g. $\boldsymbol{A}$ is a two dimensional matrix of $m \times n$ components and is defined as

$$\boldsymbol{A} = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{bmatrix}. \tag{2.4}$$

If $\boldsymbol{B}$ is an $n \times n$ square matrix and is invertible, the *inverse* of matrix $\boldsymbol{B}$, represented as $\boldsymbol{B}^{-1}$, is defined as

$$\boldsymbol{B}\boldsymbol{B}^{-1} = \boldsymbol{I}, \tag{2.5}$$

where $\boldsymbol{I}$ is an $n \times n$ square matrix known as the *identity matrix*, with 1's on the diagonal and 0's off the diagonal, defined as

$$\boldsymbol{I} = \begin{bmatrix} 1 & 0 & \cdots & 0 \\ 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 1 \end{bmatrix}. \tag{2.6}$$

### 2.1.2 Vector Spaces

A set of vectors $\boldsymbol{v} = \bar{\boldsymbol{v}}_0, \bar{\boldsymbol{v}}_1, \cdots, \bar{\boldsymbol{v}}_k$ in $\mathbb{R}^n$ is called a *vector space*, or *subspace*, in $\mathbb{R}^n$. The vectors of set $\boldsymbol{v}$ are said to be *linearly independent* if there exist no scalar constants $C_1$, $C_0, \cdots, C_k$ that satisfy the dependence relation

$$\bar{\boldsymbol{v}}_0 = C_1\bar{\boldsymbol{v}}_1 + \cdots + C_k\bar{\boldsymbol{v}}_k, \tag{2.7}$$

that is, no vector can be expressed as a linear combination of the other vectors in the set [30]. In this special case it is said that the span of the vectors $\bar{\boldsymbol{v}}_0, \bar{\boldsymbol{v}}_1, \cdots, \bar{\boldsymbol{v}}_k$ form a basis for the subspace $\boldsymbol{v}$ within the space $\mathbb{R}^n$. If the subset $\boldsymbol{v}$ forms the columns of matrix $\boldsymbol{V}$ such that

$$\boldsymbol{V} = \begin{bmatrix} \bar{\boldsymbol{v}}_0 & \bar{\boldsymbol{v}}_1 & \cdots & \bar{\boldsymbol{v}}_k \end{bmatrix}, \tag{2.8}$$

then the columns of matrix $\boldsymbol{V}$ are also said to be linearly independent.

The *dot product*, or the *inner product*, of two column vectors $\bar{\boldsymbol{x}}$ and $\bar{\boldsymbol{y}}$ is defined as

$$\bar{\boldsymbol{x}} \bullet \bar{\boldsymbol{y}} = \bar{\boldsymbol{x}}^T \bar{\boldsymbol{y}} = \sum_{i=1}^{n} x_i y_i = x_1 y_1 + x_2 y_2 + \cdots + x_n y_n = \|\bar{\boldsymbol{x}}\| \|\bar{\boldsymbol{y}}\| \cos\gamma, \tag{2.9}$$

where $\gamma$ is the angle between the two vectors [30]. The dot product of two identical vectors is 1. The dot product of two orthogonal vectors is 0. A vector space where dot-products are defined is referred to as a *dot product space*.

### 2.1.3 Lines, Planes, and Hyperplanes

Machine learning based systems, such as SVMs, are often employed to classify data into separate and distinct classes. Examples of this will be presented in Chapter 3. One simple and intuitive mechanism for classification of two distinct classes is to draw a straight line through a set of data points and label the points according to whether they lie above or below the line. This strategy also applies to higher dimensional data sets by employing a plane or hyperplane in place of a line.

A straight line in $\mathbb{R}^2$ that crosses through the origin is defined as

$$mx_1 + x_2 = 0. \tag{2.10}$$

Equation 2.10 can be rewritten as a dot product of vectors $\bar{\boldsymbol{w}} = \{m, 1\}$ and $\bar{\boldsymbol{x}} = \{x_1, x_2\}$

$$\bar{\boldsymbol{w}} \bullet \bar{\boldsymbol{x}} = 0. \tag{2.11}$$

Interpreting Eq. 2.11 geometrically, vector $\bar{\boldsymbol{x}}$ is a position vector for any point $\{x_1, x_2\}$ that lies on, and is parallel to, the line $\bar{\boldsymbol{w}} \bullet \bar{\boldsymbol{x}} = 0$ [31]. Vector $\bar{\boldsymbol{w}}$ is known as the *normal vector* and is orthogonal to the line $\bar{\boldsymbol{w}} \bullet \bar{\boldsymbol{x}} = 0$, as shown in Fig. 2.1.

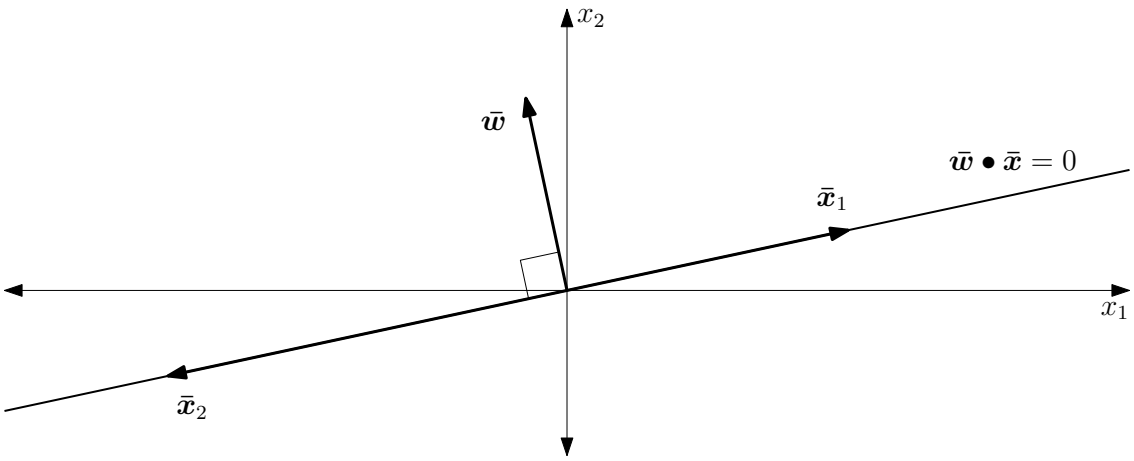



***Figure 2.1:*** *Line in $\mathbb{R}^2$ expressed as a dot product, $\bar{\boldsymbol{w}} \bullet \bar{\boldsymbol{x}} = 0$, the orthogonal normal vector $\bar{\boldsymbol{w}}$, and two possible position vectors, $\bar{\boldsymbol{x}}_{\mathbf{1}}$ and $\bar{\boldsymbol{x}}_{\mathbf{2}}$, of which both lie on, and are orthogonal to, the line.*

Similarly, a straight line in $\mathbb{R}^2$ with an offset $b$ is defined as

$$\bar{\boldsymbol{w}} \bullet \bar{\boldsymbol{x}} = b. \tag{2.12}$$

The normal vector $\bar{\boldsymbol{w}}$ is still orthogonal to the line $\bar{\boldsymbol{w}} \bullet \bar{\boldsymbol{x}} = b$, but is no longer orthogonal to the position vector $\bar{\boldsymbol{x}}$. Figure 2.2 illustrates the introduction of the offset term $b$ on the the vectors $\bar{\boldsymbol{w}}$ and $\bar{\boldsymbol{x}}$, and line itself.

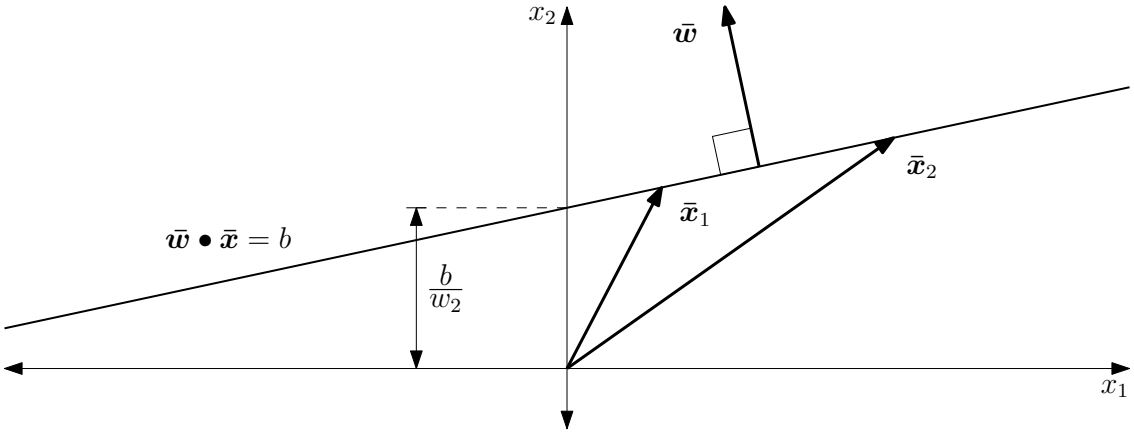



***Figure 2.2:*** *Line in $\mathbb{R}^2$ expressed as a dot product, $\bar{\boldsymbol{w}} \bullet \bar{\boldsymbol{x}} = b$, the orthogonal normal vector $\bar{\boldsymbol{w}}$, and two possible position vectors, $\bar{\boldsymbol{x}}_{\mathbf{1}}$ and $\bar{\boldsymbol{x}}_{\mathbf{2}}$, of which both are points on the line.*

This theory can also be extended to apply to planes in $\mathbb{R}^3$ and hyperplanes in higher dimensional cases of $\mathbb{R}^n$ [31]. Figure 2.3 shows a plane in $\mathbb{R}^3$ defined by the dot product $\bar{\boldsymbol{w}} \bullet \bar{\boldsymbol{x}} = b$.

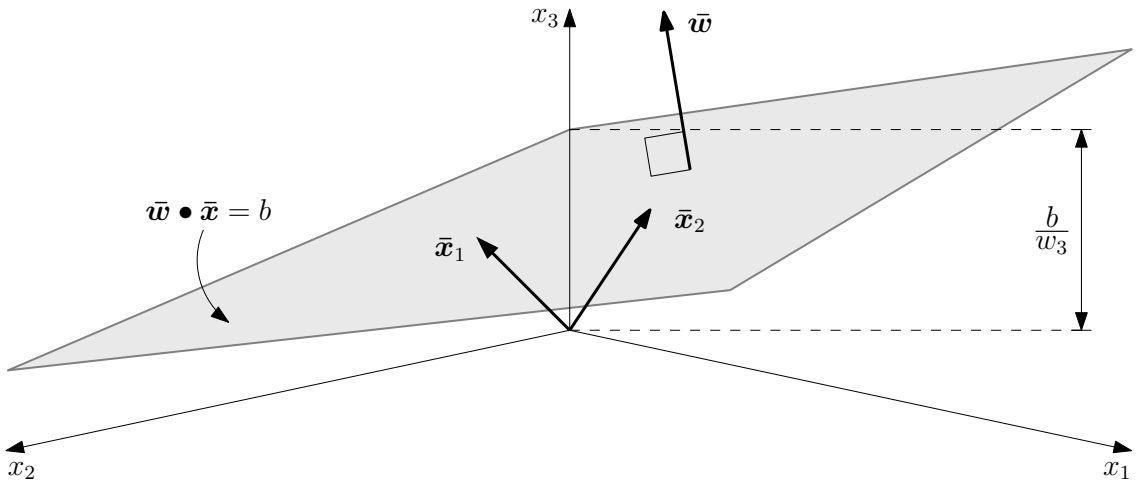



***Figure 2.3:*** *Plane in $\mathbb{R}^3$ expressed as a dot product, $\bar{\boldsymbol{w}} \bullet \bar{\boldsymbol{x}} = b$, the orthogonal normal vector $\bar{\boldsymbol{w}}$, and two possible position vectors, $\bar{\boldsymbol{x}_1}$ and $\bar{\boldsymbol{x}_2}$, of which both are points on the plane.*

## 2.2 Optimisation Problems

Optimisation problems involve finding the best solution for some *objective function* $\phi : \mathbb{R}^n \to \mathbb{R}$ given some *constraint* $h_i : \mathbb{R}^n \to \mathbb{R}$ with *bound* $c_i$. An optimisation problem is formally defined as

$$\min_{\bar{\boldsymbol{x}}} \phi(\bar{\boldsymbol{x}}), \tag{2.13}$$

such that

$$h_i(\bar{\boldsymbol{x}}) \geq c_i, \tag{2.14}$$

where $i = 1, \cdots, k$ for all $\bar{\boldsymbol{x}} \in \mathbb{R}^n$. Any value $\bar{\boldsymbol{x}} \in \mathbb{R}^n$ that satisfies the constraints is called a *feasible solution*. The optimisation aims to find the feasible solution $\bar{\boldsymbol{x}}^*$ that minimises the objective function such that for any other feasible solution $\bar{\boldsymbol{q}} \in \mathbb{R}^n$ the equality

$$\phi(\bar{\boldsymbol{x}}^*) \leq \phi(\bar{\boldsymbol{q}}) \tag{2.15}$$

holds [32]. Optimisation problems are not limited to just objective function minimisation. The following identities provide a mechanism to perform objective function maximisation:

$$\max \phi(\bar{\boldsymbol{x}}) = |\min -\phi(\bar{\boldsymbol{x}})|, \tag{2.16}$$

$$\max \phi(\bar{\boldsymbol{x}}) = \min \frac{1}{\phi(\bar{\boldsymbol{x}})}, \tag{2.17}$$

provided $1/\phi(\bar{\boldsymbol{x}})$ is defined. An optimisation problem is *linear* when the associated objective function and constraints are linear. When either the objective function or the constraints are not linear, the optimisation problem is *non-linear*.

The optimisation problems encountered in SVM training are non-linear and known specifically as *convex optimisation problems*. A convex optimisation problem has a convex ob-

jective function and linear constraints. Consider the function $f : \mathbb{R} \to \mathbb{R}$ shown in Fig. 2.4. Let $a, b \in \mathbb{R}$ be any values such that $a < b$, and let $g : \mathbb{R} \to \mathbb{R}$ be a linear function such that $g(a) = f(a)$ and $g(b) = f(b)$. The function $f$ is said to be convex if $f(x) \leq g(x)$ for all values $x \in \mathbb{R}$ such that $a < x < b$. For the convex function $f$, the line $g$ can touch the graph of the function for any interval $[a, b]$ in $\mathbb{R}$, but cannot cross it [31].

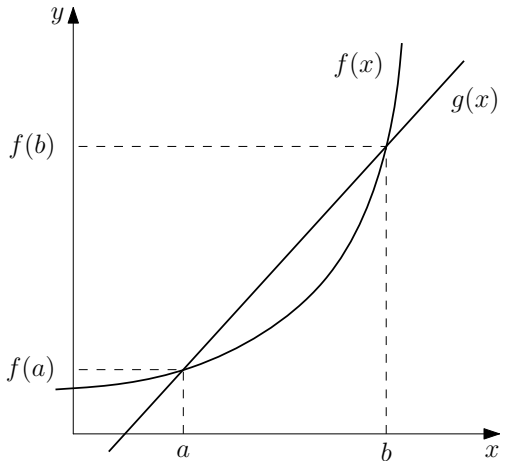



***Figure 2.4:*** *The linear function $g(x)$ intersects the convex function $f(x)$ at points $(a, f(a))$ and $(b, f(b))$.*

One method of solving convex optimisation problems is via *quadratic programming.*

## 2.3 Taylor Series

It is often computationally necessary to express known functions as an infinite series known as a *power series.* A *Taylor series* is an infinite sum approximation of a function with terms calculated from the function's derivatives at a single point. For example the exponential function $e^x$ can be expressed as the power series

$$e^x = \sum_{n=0}^{\infty} \frac{x^n}{n!} = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \frac{x^4}{4!} + \cdots , \tag{2.18}$$

and, the hyperbolic tangent function $\tanh(x)$ can be approximated by the Taylor series

$$\tanh(x) = x - \frac{x^3}{3} + \frac{2x^5}{15} - \frac{17x^7}{315} + \cdots , |x| < \frac{\pi}{2}. \tag{2.19}$$

## 2.4 State-Space Methods

*State-Space* methods provide a powerful time-domain mechanism, as opposed to classical Laplacian frequency-domain techniques, for the design and analysis of digital systems and the dataspaces the multidimentional mathematics occupy. State-space methods exploit multidimensional mathematics of linear algebra. Thus the analysis of both Single-Input-Single-Output, Multiple-Input-Single-Output, and Multiple-Input-Multiple-Output systems with many free variables is easily undertaken, manageable, and well suited to digital

and computational investigation, analysis, and control [33].

The state-space of some system, with $p$ inputs, $q$ outputs, and $n$ state variables, is defined by the following two equations

$$\dot{\bar{\boldsymbol{x}}} = \boldsymbol{A}\bar{\boldsymbol{x}} + \boldsymbol{B}\bar{\boldsymbol{u}}, \tag{2.20}$$

$$\bar{\boldsymbol{y}} = \boldsymbol{C}\bar{\boldsymbol{x}} + \boldsymbol{D}\bar{\boldsymbol{u}}, \tag{2.21}$$

where $\bar{\boldsymbol{x}} \in \mathbb{R}^n$ is the state vector, $\bar{\boldsymbol{y}} \in \mathbb{R}^q$ is the output vector, $\bar{\boldsymbol{u}} \in \mathbb{R}^p$ is the input vector, $\boldsymbol{A}$ is the $n \times n$ state matrix, $\boldsymbol{B}$ is the $n \times p$ input matrix, $\boldsymbol{C}$ is the $q \times n$ output matrix, $\boldsymbol{D}$ is a $q \times p$ matrix and is known as the feedforward matrix, and $\dot{\bar{\boldsymbol{x}}} \in \mathbb{R}^n$ is the next-state vector [34]. The state matrix $\boldsymbol{A}$ is a matrix representation of the system of differential equations or difference equations that describe the given system's dynamics. Knowing the entire state of a system $\bar{\boldsymbol{x}}$ at some time, and if the system's dynamics $\boldsymbol{A}$ are also known, the complete set of all states of the system's response can be computed for some input $\bar{\boldsymbol{u}}$. By modifying the $\boldsymbol{B}$ and $\boldsymbol{D}$ matrices accordingly some desired form of system control can also be implemented [35].

The *Gradient* $\nabla$ [36], [37] is a calculus-based tool useful for determining properties of some state-space, described by the state matrix $\boldsymbol{A}$, and of specific regions of state-space, identified by points defined by the state vector and next state vector $\bar{\boldsymbol{x}}$ and $\dot{\bar{\boldsymbol{x}}}$ respectively.

The Gradient $\nabla$ of state matrix $\boldsymbol{A}$ column $A_i$ is the $n \times 1$ column-vector of first-order partial derivatives of the time-evolution function described by column $A_i$ with respect to $\bar{\boldsymbol{x}}$, thus

$$\nabla_{\bar{\boldsymbol{x}}} A_i = \nabla A_i = \frac{\partial A_i}{\partial \bar{\boldsymbol{x}}} = \begin{bmatrix} \dfrac{\partial A_i}{\partial x_1} & \cdots & \dfrac{\partial A_i}{\partial x_n} \end{bmatrix}. \tag{2.22}$$

# Chapter 3

# Literature Review

This chapter provides a summary of the core texts pertaining to the scope of this work. It provides a complete overview of all relevant texts considered as part of this research work. This chapter identifies key concepts and gaps in the literature that this research work aims to address, supplement, and extend where appropriate.

The chapter is organised into the three sections; Machine Learning with Support Vector Machines, Digital Signal Processing and Altera Stratix V FPGA, and finally, Chaotic and Nonlinear Systems.

## 3.1 Machine Learning with Support Vector Machines

Machine learning as a field is a very vast and complex one; to provide even an overview of all machine learning paradigms would require one very large volume, if not more, just to introduce the core fundamentals of each. With this in mind, this section has been written with the intend in providing only the concepts pertaining to Support Vector Machines explicitly, and where appropriate, the theories and literature that provide a solid foundation for the continued research and development towards implementing a real-time hardware-based SVM system.

### 3.1.1 Maximum-Margin Classifiers and SVMs

Considerable work has gone into SVM research and development, and a clear evolution of the paradigm can be observed in the literature. This subsection introduces the SVM paradigm from the its Maximum-Margin Classifiers ancestry through to Vapnik's influential and relevant theory of statistical-based machine learning and the SVMs advanced learning, classifying, and regression applications.

#### 3.1.1.1 Maximum-Margin Classifiers

Before providing an exposition of the Maximum-Margin Classifier paradigm, several related concepts require formal definition. In binary classification problems a hyperplane *supports* a class of data-points if it is parallel to a linear decision surface and all points

of its respective class are either above or below it. Such a hyperplane is referred to as a *supporting hyperplane*. The distance between the two supporting hyperplanes is called a *margin*. A decision surface is *optimal* if it is equidistant from the two supporting hyperplanes and maximises their margin [31]. Figure 3.1 provides an illustration of these concepts in $\mathbb{R}^2$.

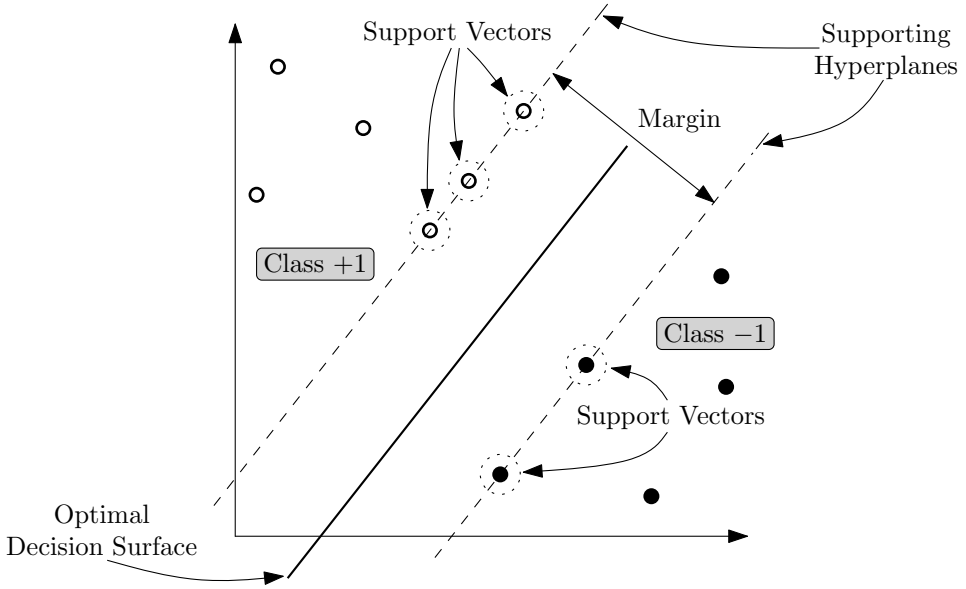



***Figure 3.1:*** *Optimal decision surface with its two supporting hyperplanes separating two linearly separable classes.*

Given a linearly separable training data set D

$$D = \{(\bar{\boldsymbol{x}}_1, y_1), (\bar{\boldsymbol{x}}_2, y_2), \cdots, (\bar{\boldsymbol{x}}_k, y_k)\} \subseteq \mathbb{R}^n \times \{+1, -1\}, \tag{3.1}$$

one can compute a maximum-margin decision surface $\bar{\boldsymbol{w}}^* \bullet \bar{\boldsymbol{x}} = b^*$ with the optimisation

$$\min \phi(\bar{\boldsymbol{w}}, b) = \min_{\bar{\boldsymbol{w}}, b} \tfrac{1}{2} \bar{\boldsymbol{w}} \bullet \bar{\boldsymbol{w}} \tag{3.2}$$

subject to the constraints

$$\bar{\boldsymbol{w}} \bullet (y_i \bar{\boldsymbol{x}}_i) \geq 1 + y_i b \qquad \text{for all} \quad (\bar{\boldsymbol{x}}_i, y_i) \in D. \tag{3.3}$$

Figure 3.2 illustrates an example of a maximum-margin class separation of some arbitrary linearly separable data set in $\mathbb{R}^2$ by an optimal decision surface $\bar{\boldsymbol{w}}^* \bullet \bar{\boldsymbol{x}} = b^*$. Corresponding supporting hyperplanes are separated by an optimal margin and are both equidistant from the optimal decision surface.

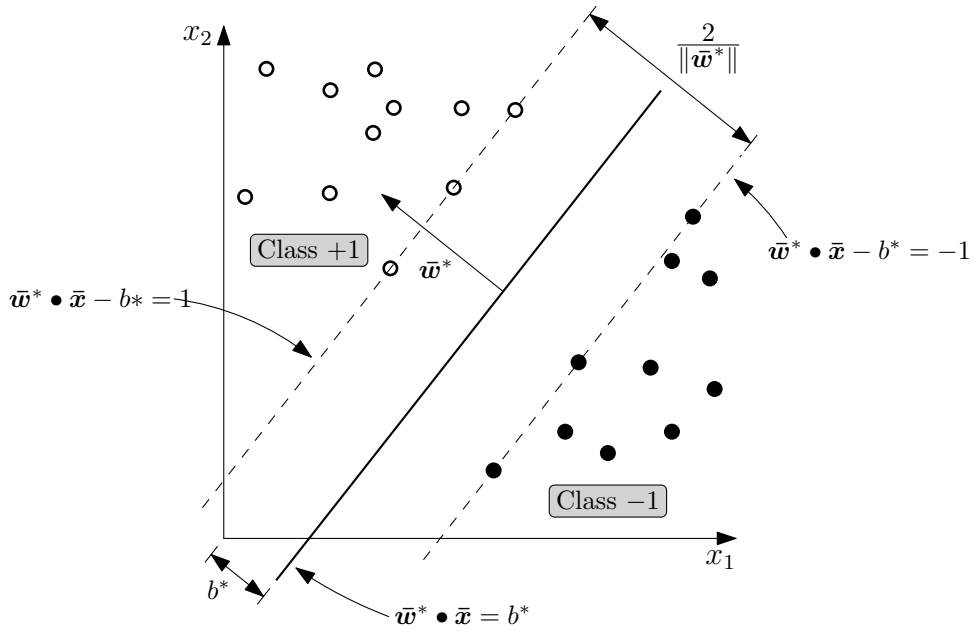



***Figure 3.2:*** *Maximum-Margin class separation example.*

Quadratic Programming techniques are used to optimise the margin. The objective function shown in Equation 3.2 is a convex function

$$\phi(\bar{\boldsymbol{w}}, b) = \frac{1}{2}\bar{\boldsymbol{w}} \bullet \bar{\boldsymbol{w}} = \frac{1}{2}(w_1^2 + \cdots + w_n^2), \tag{3.4}$$

where $\bar{\boldsymbol{w}} = (w_1, \cdots, w_n)$. The objective function $\frac{1}{2}\bar{\boldsymbol{w}} \bullet \bar{\boldsymbol{w}}$ for two-dimensional space $\mathbb{R}^2$ is shown in Fig. 3.3.

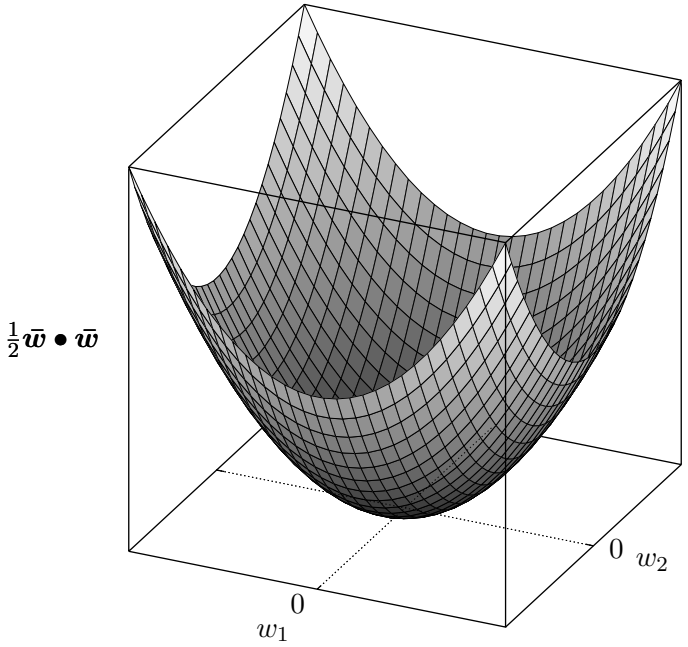



***Figure 3.3:*** *Objective function* $\frac{1}{2}\bar{\boldsymbol{w}} \bullet \bar{\boldsymbol{w}}$ *in* $\mathbb{R}^2$.

Convexity implies it is possible to find a global minimum for the objective function $\phi(\bar{\boldsymbol{w}}, b)$.

A quadratic program takes the form

$$\bar{\boldsymbol{w}}^* = \arg\min_{\bar{\boldsymbol{w}}} \left( \frac{1}{2} \bar{\boldsymbol{w}}^T \boldsymbol{Q} \bar{\boldsymbol{w}} - \bar{\boldsymbol{q}} \bullet \bar{\boldsymbol{w}} \right), \tag{3.5}$$

subject to the constraints

$$\boldsymbol{X}^T \bar{\boldsymbol{w}} \geq \bar{\boldsymbol{c}}. \tag{3.6}$$

As before, $\boldsymbol{Q}$ is an $n \times n$ matrix, $\boldsymbol{X}$ is an $n \times k$ matrix, $\bar{\boldsymbol{w}}^*$, $\bar{\boldsymbol{w}}$, and $\bar{\boldsymbol{q}}$ are $n$-dimensional vectors, and $\bar{\boldsymbol{c}}$ is an $k$-dimensional vector. By letting $\boldsymbol{Q}$ be the identity matrix $\boldsymbol{I}$ and $\bar{\boldsymbol{q}}$ be the zero vector $\bar{\boldsymbol{0}}$, the general optimisation problem can be transformed into a quadratic program compatible with the objective function shown in Equation 3.4:

$$\bar{\boldsymbol{w}}^* = \arg\min_{\bar{\boldsymbol{w}}} \left( \frac{1}{2} \bar{\boldsymbol{w}}^T \boldsymbol{I} \bar{\boldsymbol{w}} - \bar{\boldsymbol{0}} \bullet \bar{\boldsymbol{w}} \right) = \arg\min_{\bar{\boldsymbol{w}}} \left( \frac{1}{2} \bar{\boldsymbol{w}} \bullet \bar{\boldsymbol{w}} \right). \tag{3.7}$$

where $\bar{\boldsymbol{w}}^T \boldsymbol{I} \bar{\boldsymbol{w}} = \bar{\boldsymbol{w}} \bullet \bar{\boldsymbol{w}}$. The constraints shown in Equation 3.3 can be rewritten in a form compatible with the quadratic program constraints shown in Equation 3.6,

$$(y_i \bar{\boldsymbol{x}}_i) \bullet \bar{\boldsymbol{w}} \geq 1 + y_i b \iff \boldsymbol{X}^T \bar{\boldsymbol{w}} \geq \bar{\boldsymbol{c}} \tag{3.8}$$

for all $(y_i \bar{\boldsymbol{x}}_i) \in D$ with $i = 1, \cdots, k$ and $\bar{\boldsymbol{x}}_i = (x_{i_1}, x_{i_2}, \cdots, x_{i_n})$. Thus the $\boldsymbol{X}$ matrix takes the form

$$\boldsymbol{X} = \begin{bmatrix} y_1 x_{1_1} & y_2 x_{2_1} & \cdots & y_k x_{k_1} \\ y_1 x_{1_2} & y_2 a_{2_2} & \cdots & y_k x_{k_2} \\ \vdots & \vdots & \ddots & \vdots \\ y_1 x_{1_n} & y_2 x_{2_n} & \cdots & y_k x_{k_n} \end{bmatrix}. \tag{3.9}$$

Similarly the $\bar{\boldsymbol{c}}$ vector takes the form

$$\bar{\boldsymbol{c}} = \begin{bmatrix} 1 + y_1 b \\ 1 + y_2 b \\ \vdots \\ 1 + y_k b \end{bmatrix}. \tag{3.10}$$

The algorithm shown in Listing 3.1 shows how a decision surface with a maximum margin is computed using a standard quadratic program solver [31]. The quantity $r$ represents the radius of the training set $D$. The constant $q$ defines the size of the search interval for offset term $b$ values; here it is set to 1000.

***Listing 3.1:*** *Quadratic programming algorithm*

```
1  let D = {(x̄_1, y_1), (x̄_2, y_2), ··· , (x̄_k, y_k)} ⊆ ℝ^n × {+1, −1}
2  r ← max{‖x̄‖ | (x̄, y) ∈ D}
3  q ← 1000
4  let w̄* and b* be undefined
5  construct X according to Equation 3.9 using D
6  for each b ∈ [−q, q] do
7      construct c̄ according to Equation 3.10 using b
8      w̄ ← solve(I, 0̄, X, c̄)
```

```
    if (w̄ is defined and w̄* is undefined) or (w̄ is defined and ‖w̄‖ < ‖w̄*‖) then
        w̄* ← w̄
        b* ← b
    end if
end for
if w̄* is undefined then
    stop constraints not satisfiable
else if ‖w̄*‖ > q/r then
    stop bounding assumption of ‖w̄‖ violated
end if
return (w̄*, b*)
```

#### 3.1.1.2 Support Vector Machine Classifiers

A Support Vector Machine (SVM) is the optimisation of the Lagrangian dual of the Maximum-Margin Classifier. Consider the Lagrangian $L(\alpha, x)$ defined as

$$L(\alpha, x) = \frac{1}{2}x^2 - \alpha(x - 2). \tag{3.11}$$

Therefore the optimisation of the Lagrangian convex objective function $L(\alpha, x)$, as shown in Fig. 3.4 in $\mathbb{R}^2$, is a unique saddle-point on $L(\alpha, x)$.

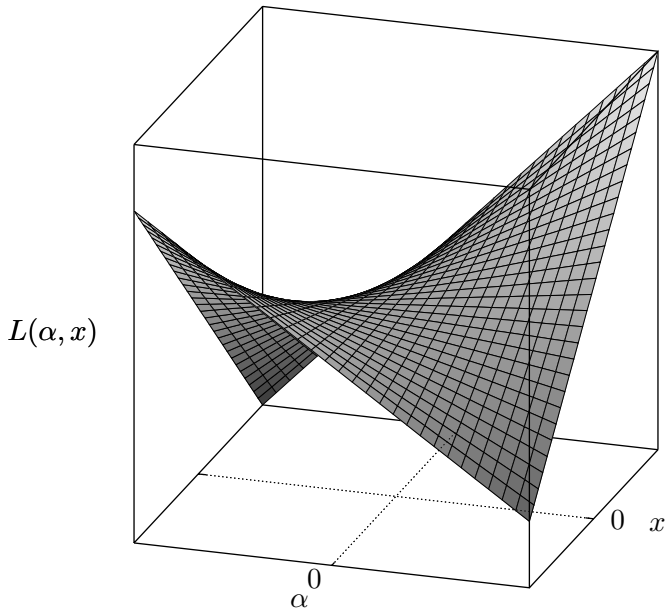



***Figure 3.4:*** *Lagrangian objective function* $L(\alpha, x) = \frac{1}{2}x^2 - \alpha(x - 2)$ *in* $\mathbb{R}^2$.

The optimal unique saddle point on the Lagrangian $L(\alpha, x)$ has to occur where the the gradient of $L(\alpha, x)$ with respect to $x$ is equal to zero:

$$\frac{\partial L(\alpha, x^*)}{\partial x^*} = x^* - \alpha = 0 \tag{3.12}$$

where $x^*$ represents the value that minimises the Lagrangian with respect to $x$ at the saddle-point. Solving for $x^*$ and substituting into Eq. 3.11 gives the Lagrangian dual

optimisation with $\phi'(\alpha) = L(\alpha, x^*)$,

$$\arg\max_{\alpha} \phi'(\alpha) = \arg\max_{\alpha} \left(2\alpha - \frac{1}{2}\alpha^2\right) \tag{3.13}$$

subject to

$$\alpha =\geq 0. \tag{3.14}$$

The derivation of the maximum-margin classifier Lagrangian dual optimisation problem, the SVM, follows. Recall the maximum-margin optimisation problem

$$\arg\min_{\bar{\boldsymbol{w}},b} \phi(\bar{\boldsymbol{w}}, b) = \arg\min_{\bar{\boldsymbol{w}},b} \left(\frac{1}{2}\bar{\boldsymbol{w}} \bullet \bar{\boldsymbol{w}}\right) \tag{3.15}$$

subject to the constraints

$$g_i(\bar{\boldsymbol{w}}, b) = y_i(\bar{\boldsymbol{w}} \bullet \bar{\boldsymbol{x}}_i - b) - 1 \geq 0 \tag{3.16}$$

for $i = 1, \cdots, k$. The corresponding Lagrangian is constructed

$$\begin{aligned} L(\bar{\boldsymbol{\alpha}}, \bar{\boldsymbol{w}}, b) &= \phi(\bar{\boldsymbol{w}}, b) - \sum_{i=1}^{k} \alpha_i g_i(\bar{\boldsymbol{w}}, b) \\ &= \frac{1}{2}\bar{\boldsymbol{w}} \bullet \bar{\boldsymbol{w}} - \sum_{i=1}^{k} \alpha_i (y_i(\bar{\boldsymbol{w}} \bullet \bar{\boldsymbol{x}}_i - b) - 1) \\ &= \frac{1}{2}\bar{\boldsymbol{w}} \bullet \bar{\boldsymbol{w}} - \sum_{i=1}^{k} \alpha_i y_i \bar{\boldsymbol{w}} \bullet \bar{\boldsymbol{x}}_i + b \sum_{i=1}^{k} \alpha_i y_i + \sum_{i=1}^{k} \alpha_i. \end{aligned} \tag{3.17}$$

Thus the Lagrangian optimisation problem for maximum-margin classifiers is given by

$$\max_{\alpha} \min_{\bar{\boldsymbol{w}},b} L(\bar{\boldsymbol{\alpha}}, \bar{\boldsymbol{w}}, b), \tag{3.18}$$

subject to

$$\alpha_i =\geq 0. \tag{3.19}$$

for $i = 1, \cdots, k$. Let $\bar{\boldsymbol{\alpha}}^*, \bar{\boldsymbol{w}}^*$, and $b^*$ be a solution to the Lagrangian optimisation problem such that

$$\max_{\alpha} \min_{\bar{\boldsymbol{w}},b} L(\bar{\boldsymbol{\alpha}}, \bar{\boldsymbol{w}}, b) = L(\bar{\boldsymbol{\alpha}}^*, \bar{\boldsymbol{w}}^*, b^*), \tag{3.20}$$

Since $\phi$ is convex and the constraints $g_i$ are linear, the solution $\bar{\boldsymbol{\alpha}}^*, \bar{\boldsymbol{w}}^*$, and $b^*$ will satisfy

the following Karush-Kuhn-Tucker (KKT) conditions:

$$\frac{\partial L(\bar{\boldsymbol{\alpha}}^*, \bar{\boldsymbol{w}}^*, b^*)}{\partial \bar{\boldsymbol{w}}} = \bar{\mathbf{0}}, \tag{3.21}$$

$$\frac{\partial L(\bar{\boldsymbol{\alpha}}^*, \bar{\boldsymbol{w}}^*, b^*)}{\partial b} = 0, \tag{3.22}$$

$$\alpha_i^*(y_i(\bar{\boldsymbol{w}}^* \bullet \bar{\boldsymbol{x}}_i - b^*) - 1) = 0, \tag{3.23}$$

$$y_i(\bar{\boldsymbol{w}}^* \bullet \bar{\boldsymbol{x}}_i - b^*) - 1 \geq 0, \tag{3.24}$$

$$\alpha_i^* \geq 0 \tag{3.25}$$

for $i = 1, \cdots, k$. By taking the partial derivatives of the Lagrangian defined in Eq. 3.17 with respect to $\bar{\boldsymbol{w}}$ and $b$, setting each to zero as defined in the KKT conditions defined in Eq. 3.21 and Eq. 3.22, and substituting back into Eq. 3.17 the Lagrangian dual for maximum-margin classifiers can be constructed

$$\phi'(\bar{\boldsymbol{\alpha}}) = L(\bar{\boldsymbol{\alpha}}, \bar{\boldsymbol{w}}^*, b^*) = \sum_{i=1}^{k} \alpha_i - \frac{1}{2} \sum_{i=1}^{k} \sum_{j=1}^{k} \alpha_i \alpha_j y_i y_j \bar{\boldsymbol{x}}_i \bullet \bar{\boldsymbol{x}}_j. \tag{3.26}$$

Thus the SVM is defined as follows. Given a labelled linearly separable training data set D defined as

$$D = \{(\bar{\boldsymbol{x}}_1, y_1), (\bar{\boldsymbol{x}}_2, y_2), \cdots, (\bar{\boldsymbol{x}}_k, y_k)\} \subseteq \mathbb{R}^n \times \{+1, -1\}, \tag{3.27}$$

where $y_i = f(\bar{\boldsymbol{x}}_i)$ and $f$ is some target function where $f : \mathbb{R}^n \to \{+1, -1\}$, one can compute a model $\hat{f} : \mathbb{R}^n \to \{+1, -1\}$ using D such that

$$\hat{f}(\bar{\boldsymbol{x}}) \cong f(\bar{\boldsymbol{x}}) \tag{3.28}$$

for all $\bar{\boldsymbol{x}} \in \mathbb{R}^n$. Therefore support vector models are trained with the dual Lagrangian optimisation for maximum-margin classifiers

$$\bar{\boldsymbol{\alpha}}^* = \arg\max_{\bar{\boldsymbol{\alpha}}} \left( \sum_{i=1}^{k} \alpha_i - \frac{1}{2} \sum_{i=1}^{k} \sum_{j=1}^{k} \alpha_i \alpha_j y_i y_j \bar{\boldsymbol{x}}_i \bullet \bar{\boldsymbol{x}}_j \right), \tag{3.29}$$

subject to the constraints

$$\sum_{i=1}^{k} \alpha_i y_i = 0, \text{ and} \tag{3.30}$$

$$\alpha_i \geq 0, \tag{3.31}$$

where $\bar{\boldsymbol{\alpha}} = \{\alpha_1, \alpha_2, \cdots, \alpha_k\}$ are *Lagrangian multipliers* and $i = 1, \cdots, k$. Data points with non-zero Lagrangian multipliers are called *support vectors* $\bar{\boldsymbol{x}}_{sv}$.

The model $\hat{f}(\bar{\boldsymbol{x}})$ is defined as the linear support vector machine

$$\hat{f}(\bar{\boldsymbol{x}}) = \quad \operatorname{sgn}\left(\sum_{i=1}^{k} \alpha_i^* y_i \bar{\boldsymbol{x}}_i \bullet \bar{\boldsymbol{x}} - \sum_{i=1}^{k} \alpha_i^* y_i \bar{\boldsymbol{x}}_i \bullet \bar{\boldsymbol{x}}_{sv^+} + 1\right), \tag{3.32}$$

where one support vector $\bar{\boldsymbol{x}}_{sv^+}$ is chosen from from the set of available support vectors,

$$(\bar{\boldsymbol{x}}_{sv^+}, +1) \in \{(\bar{\boldsymbol{x}}_i, +1) \mid (\bar{\boldsymbol{x}}_i, +1) \in D \ \text{ and } \ \alpha_i^* > 0\}. \tag{3.33}$$

In practice data sets are rarely linearly separable. This can be overcome by the application of *kernel functions*. An appropriately chosen kernel function will transform the *input space* where the data set is is not linearly separable to a higher-dimensional space called a *feature space* where the data set is linearly separable. Conversely this kernel function mapping transforms a nonlinear decision problem in the input space into a linear decision problem in the feature space. Also by choosing the right kernel function all calculations can be performed in the input space, implying that the act of performing the transformations is completely avoided, thus so too is any computational cost incurred by the transformations. This is called the *kernel trick*. Let $\Phi : \mathbb{R}^n \to \mathbb{R}^n$ be the identity function on $\mathbb{R}^n$, then the kernel function $\psi$ can be defined as

$$\psi(\bar{\boldsymbol{x}}, \bar{\boldsymbol{y}}) = \Phi(\bar{\boldsymbol{x}}) \bullet \Phi(\bar{\boldsymbol{y}}) = \bar{\boldsymbol{x}} \bullet \bar{\boldsymbol{y}}, \tag{3.34}$$

where $\bar{\boldsymbol{x}}$, $\bar{\boldsymbol{y}} \in \mathbb{R}^n$. This is the linear kernel, and the feature space is the same as the input space. The support vector models can now be trained by rewriting Equation 3.29 and optimising the Lagrangian

$$\bar{\boldsymbol{\alpha}}^* = \arg\max_{\bar{\boldsymbol{\alpha}}} \left(\sum_{i=1}^{k} \alpha_i - \frac{1}{2} \sum_{i=1}^{k} \sum_{j=1}^{k} \alpha_i \alpha_j y_i y_j \psi(\bar{\boldsymbol{x}}_i, \bar{\boldsymbol{x}}_j)\right), \tag{3.35}$$

subject to the same constraints given in Equations 3.30 and 3.31. Likewise the linear support vector machine model $\hat{f}(\bar{\boldsymbol{x}})$ can be redefined as

$$\hat{f}(\bar{\boldsymbol{x}}) = \quad \operatorname{sgn}\left(\sum_{i=1}^{k} \alpha_i^* y_i \psi(\bar{\boldsymbol{x}}_i, \bar{\boldsymbol{x}}) - \sum_{i=1}^{k} \alpha_i^* y_i \psi(\bar{\boldsymbol{x}}_i, \bar{\boldsymbol{x}}_{sv^+}) + 1\right). \tag{3.36}$$

There exist many nonlinear kernel functions $\psi$ that can be substituted into Equations 3.35 and 3.36 allowing the effective transformation of data sets from the input space to the feature space while maintaining the computation benefits of the kernel trick. Table 3.1 presents some commonly used kernel functions.

***Table 3.1:*** *Commonly used kernel functions and their free parameters.*

| Kernel Name | Kernel Function $\psi(\bar{\boldsymbol{x}}, \bar{\boldsymbol{y}})$, $\bar{\boldsymbol{x}}$, $\bar{\boldsymbol{y}} \in \mathbb{R}^n$ | Free Parameters |
|---|---|---|
| Linear Kernel | $\psi(\bar{\boldsymbol{x}}, \bar{\boldsymbol{y}}) = \bar{\boldsymbol{x}} \bullet \bar{\boldsymbol{y}}$ | none |
| Homogeneous Polynomial Kernel | $\psi(\bar{\boldsymbol{x}}, \bar{\boldsymbol{y}}) = (\bar{\boldsymbol{x}} \bullet \bar{\boldsymbol{y}})^d$ | $d \geq 2$ |
| Nonhomogeneous Polynomial Kernel | $\psi(\bar{\boldsymbol{x}}, \bar{\boldsymbol{y}}) = (\bar{\boldsymbol{x}} \bullet \bar{\boldsymbol{y}} + c)^d$ | $d \geq 2, c > 0$ |
| Gaussian Kernel | $\psi(\bar{\boldsymbol{x}}, \bar{\boldsymbol{y}}) = e^{-(\|\bar{\boldsymbol{x}} - \bar{\boldsymbol{y}}\|^2 / 2\sigma^2)}$ | $\sigma > 0$ |
| Radial Basis Function (RBF) Kernel | $\psi(\bar{\boldsymbol{x}}, \bar{\boldsymbol{y}}) = e^{-(\gamma \|\bar{\boldsymbol{x}} - \bar{\boldsymbol{y}}\|^2)}$ | $\gamma > 0$ |
| Sigmoid / Hyperbolic Tangent Kernel | $\psi(\bar{\boldsymbol{x}}, \bar{\boldsymbol{y}}) = \tanh(\bar{\boldsymbol{x}} \bullet \bar{\boldsymbol{y}} + c)$ | $c > 0$ |

Real world data sets will also always contain some form of noise. To mitigate the effects of noisy datasets it is prudent to allow the training of an SVM to make mistakes. Allowing the training algorithm to ignore certain points in a data set, points that may be discoloured by some noise present in the data, gives rise to much simpler decision surfaces, and consequently, decision surfaces that tend to generalise better.

To reduce the impact of noisy training data the introduction of *slack variables* $\xi_j$ allow troublesome points to lie on the wrong side of their respective supporting hyperplane. The slack variables measure how much error is committed by allowing the supporting hyperplane to be unconstrained by that point [31]. Classifiers that employ slack variables in this manner are known *Soft-margin classifiers*. Soft-margin class separation is illustrated in Fig. 3.5. An upper bound on the compounded error conceded by non-zero erroneous slack variables is defined as

$$C \sum_j \xi_j, \tag{3.37}$$

where $C$ is called the *cost*. A soft-margin SVM classifier can be derived by developing the Lagrangian dual of the maximum-margin classifier with the term defined in Equation 3.37 added to the primal objective function Lagrangian. The cost term $C$ only appears in the Lagrangian dual optimisation constraints. The cost $C$ controls the trade-off between margin-size and classification error. A larger cost $C$ forces the optimisation to permit fewer non-zero erroneous slack variables, therefore a smaller margin will be found, thus increasing the cost in both complexity and potentially-lost generalisation ability.

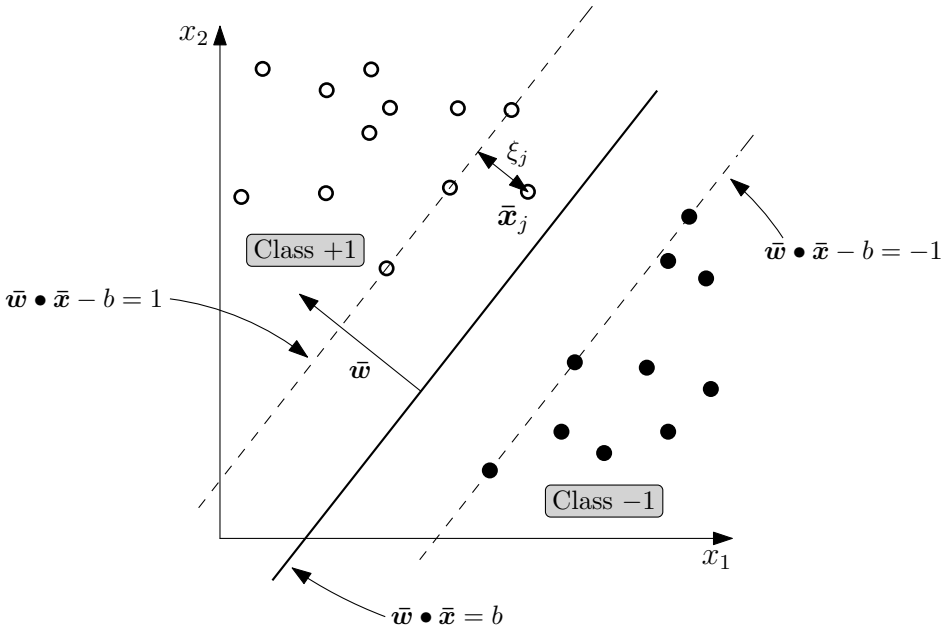



***Figure 3.5:*** *Soft-Margin Classifier class separation example.*

The soft-margin support vector models can now be trained by optimising the Lagrangian

$$\bar{\boldsymbol{\alpha}}^* = \arg\max_{\bar{\boldsymbol{\alpha}}} \left( \sum_{i=1}^{k} \alpha_i - \frac{1}{2} \sum_{i=1}^{k} \sum_{j=1}^{k} \alpha_i \alpha_j y_i y_j \psi(\bar{\boldsymbol{x}}_i, \bar{\boldsymbol{x}}_j) \right), \tag{3.38}$$

subject to the modified constraints

$$\sum_{i=1}^{k} \alpha_i y_i = 0, \text{ and} \tag{3.39}$$

$$C \geq \alpha_i \geq 0, \tag{3.40}$$

where $i = 1, \cdots, k$ and $C$ is the cost constant.

As before the support vector machine model $\hat{f}(\bar{\boldsymbol{x}})$ is defined as

$$\hat{f}(\bar{\boldsymbol{x}}) = \quad \text{sgn} \left( \sum_{i=1}^{k} \alpha_i^* y_i \psi(\bar{\boldsymbol{x}}_i, \bar{\boldsymbol{x}}) - \sum_{i=1}^{k} \alpha_i^* y_i \psi(\bar{\boldsymbol{x}}_i, \bar{\boldsymbol{x}}_{sv^+}) + 1 \right), \tag{3.41}$$

however, one must be careful to choose a support vector $\bar{\boldsymbol{x}}_{sv^+}$ with a zero valued slack variable $\xi_{sv^+}^* = 0$ from the set of available support vectors,

$$(\bar{\boldsymbol{x}}_{sv^+}, +1) \in \{(\bar{\boldsymbol{x}}_i, +1) \mid (\bar{\boldsymbol{x}}_i, +1) \in D \text{ and } 0 < \alpha_i^* < C\}. \tag{3.42}$$

#### 3.1.1.3 Statistical Learning Theory

Initially developed by Vladimir Naumovich Vapnik and Alexey Jakovlevich Chervonenkis, *Statistical Learning Theory*, or *VC-Theory*, provides the theoretical foundation that the SVM machine learning paradigm is based. At the core of this theoretical foundation lies

the *model of learning from examples*, as shown in Fig. 3.6 [26]- [28]. The model consists of three distinct elements; a random process of some unknown probability distribution $P(\bar{\boldsymbol{x}}) = P(\bar{\boldsymbol{x}}, y)$, a supervisor, and a learning machine.

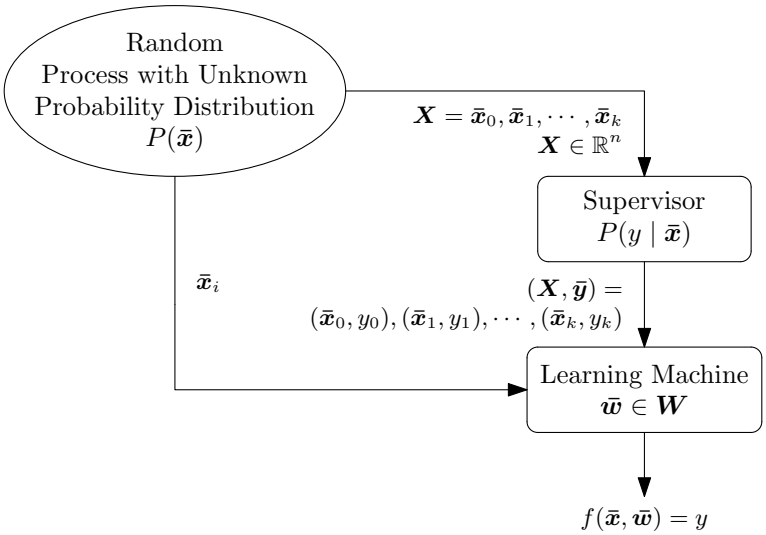



***Figure 3.6:*** *Statistical Learning Theory: The model of learning from examples.*

The random process $P(\bar{\boldsymbol{x}})$ generates independent and identically distributed (i.i.d.) data vectors $\bar{\boldsymbol{x}}_0, \cdots, \bar{\boldsymbol{x}}_k = \boldsymbol{X}$ that serve as the input to the supervisor. The supervisor determines a label $y_i \in \{-1, +1\}$ for each vector $\bar{\boldsymbol{x}}_i$ according to some unknown but fixed decision function $P(y \mid \bar{\boldsymbol{x}})$. The labelled data pairs $(\boldsymbol{X}, \bar{\boldsymbol{y}})$ serve as training data to the learning machine. Given training data, the learning machine must choose a decision function $f$ from the set of functions $F(\bar{\boldsymbol{x}}, \bar{\boldsymbol{w}}), \bar{\boldsymbol{w}} \in \boldsymbol{W}$ that best approximates the supervisor's decision function, and thus correctly predict the response $y_i$ given any vector $\bar{\boldsymbol{x}}_{\boldsymbol{i}}$ generated by the random process. This is called *the problem of learning.*

Determining the decision function or model $f \in F(\bar{\boldsymbol{x}}, \bar{\boldsymbol{w}})$ that best approximates the supervisor's unknown decision function is achieved by minimising the difference between the expected outputs $y_i$ and the response from the learning machine $f(\bar{\boldsymbol{x}}, \bar{\boldsymbol{w}})$, known as the loss function $L(y, f(\bar{\boldsymbol{x}}, \bar{\boldsymbol{w}}))$. The loss function compares training data labels and model $f$ output labels; if the model performs a classification error on a data point $x_i$ the loss function returns a 1, otherwise it returns a 0. The loss function is defined as

$$L(y, f(\bar{\boldsymbol{x}}, \bar{\boldsymbol{w}})) = \begin{cases} 0 & \text{if} \quad y = f(\bar{\boldsymbol{x}}, \bar{\boldsymbol{w}}), \\ 1 & \text{if} \quad y \neq f(\bar{\boldsymbol{x}}, \bar{\boldsymbol{w}}). \end{cases} \tag{3.43}$$

The *expected loss* or the *expected risk* of some function $f \in F(\bar{\boldsymbol{x}}, \bar{\boldsymbol{w}})$ over the entire data-space is defined as

$$R[f] = E[L(y, f(\bar{\boldsymbol{x}}, \bar{\boldsymbol{w}}))] = \int L(y, f(\bar{\boldsymbol{x}}, \bar{\boldsymbol{w}}))dP(\bar{\boldsymbol{x}}, y). \tag{3.44}$$

Thus minimising the expected risk finds the best generalisation of the supervisor's decision function, the optimal learning machine model $f^*$,

$$f^* = \underset{f \in F(\bar{\boldsymbol{x}}, \bar{\boldsymbol{w}})}{\arg\min} R[f], \tag{3.45}$$

and appears to solve the problem of learning [31]. However the joint probability distribution $P(\bar{\boldsymbol{x}}, y)$ is unknown, thus Eq. 3.45 is of no practical usefulness or benefit. The joint probability distribution of the observed training data, however, is known. The risk estimated using the training data is called the *empirical risk* $R_{emp}[f]$ of some model $f$, and is defined as

$$R_{emp}[f] = E[L(y, f(\bar{\boldsymbol{x}}, \bar{\boldsymbol{w}}))] = \frac{1}{k} \sum_{i=0}^{k-1} L(y_i, f(\bar{\boldsymbol{x}}_i, \bar{\boldsymbol{w}})), \tag{3.46}$$

where $(\bar{\boldsymbol{x}}_i, y_i) \in (\boldsymbol{X}, \bar{\boldsymbol{y}})$. Thus by minimising the empirical risk, known as *Empirical Risk Minimisation*, the optimal learning machine model $f^*$ can be calculated;

$$f^* = \underset{f \in F(\bar{\boldsymbol{x}}, \bar{\boldsymbol{w}})}{\arg\min} R_{emp}[f]. \tag{3.47}$$

Minimising the empirical risk $R_{emp}[f]$ and finding the optimum model $f^*$ from the set of all functions $F(\bar{\boldsymbol{x}}, \bar{\boldsymbol{w}})$ has the upshot of defining a model that will exhibit a risk $R_{emp}[f]$ or classification error of effectively zero for the training data. The caveat to this situation however is the optimal model $f^*$ does not generalise the supervisor's decision function well, and the model will exhibit data *overfitting* when new unseen unlabelled data is classified. This will be observed as an increase in the risk $R_{emp}[f]$ as classification errors.

To overcome this drawback of the Empirical Risk Minimisation (ERM), Vapnik developed the concept of *Structural Risk Minimisation*. Structural Risk Minimisation (SRM) utilises the concept of classifier model complexity, known as the *VC-dimension* $h$, for a given training data set [38], [39]; the VC-dimension of a model class is data dependent [31]. The VC-dimension is a measure of how well a binary classifier can model the boundary between two classes; the larger the VC-dimension, the more complex the classifier, and, the better it can separate the data into its two respective classes.

The formal definition of a classifier's VC-dimesion follows. Let $\hat{F}[\gamma]$ denote a class of linear classifiers all with the same margin size $\gamma$. Assume that the model class $\hat{F}[\gamma]$ is closed under rotation and translation, or, for all $\hat{f} \in \hat{F}[\gamma]$, all rotations $\rho$ and all translations $\tau$ have $\rho(\hat{f}) \in \hat{F}[\gamma]$ and $\tau(\hat{f}) \in \hat{F}[\gamma]$. The VC-dimension of a model class $\hat{F}[\gamma]$ defined over some data set $D$ is the size of the largest finite subset of $D$ *shattered* by $\hat{F}[\gamma]$.

The following exposition and Fig. 3.7 provides an elaboration and visual illustration respectively of the VC-dimension for two classifier model classes, $\hat{F}[\gamma_1]$ and $\hat{F}[\gamma_2]$, over a small three-vector data set $D \subset \mathbb{R}^2$. Let the two distinct classes of classifiers $\hat{F}[\gamma_1]$ and $\hat{F}[\gamma_2]$ be defined over $D$. The margin $\gamma_1$ is chosen such that the classifier model class can separate all three instances for all possible binary label assignments, as shown in Fig. 3.7

(a). In this instance the VC-dimension of $\hat{F}[\gamma_1]$ is 3, or, $h_1 = 3$. Since the VC-dimension $h_1$ is equal to the dimension of the data set, it is said that $\hat{F}[\gamma_1]$ shatters $D$. The margin $\gamma_2$ is chosen such that $\gamma_2 > \gamma_1$, and so that the class of classifiers $\hat{F}[\gamma_2]$ cannot separate all instances perfectly, as shown in Fig. 3.7 (b). However, if the grey instance in Fig. 3.7 (b) is removed, the classifiers in $\hat{F}[\gamma_2]$ can shatter this subset of $D$. Therefore the VC-dimension for the model class $\hat{F}[\gamma_2]$ is 2, or $h_2 = 2$. As $h_1 > h_2$ it is said the classifiers in $\hat{F}[\gamma_1]$ are more complex than the classifiers in $\hat{F}[\gamma_2]$, or, $\hat{F}[\gamma_1] \supset \hat{F}[\gamma_2]$ [31].

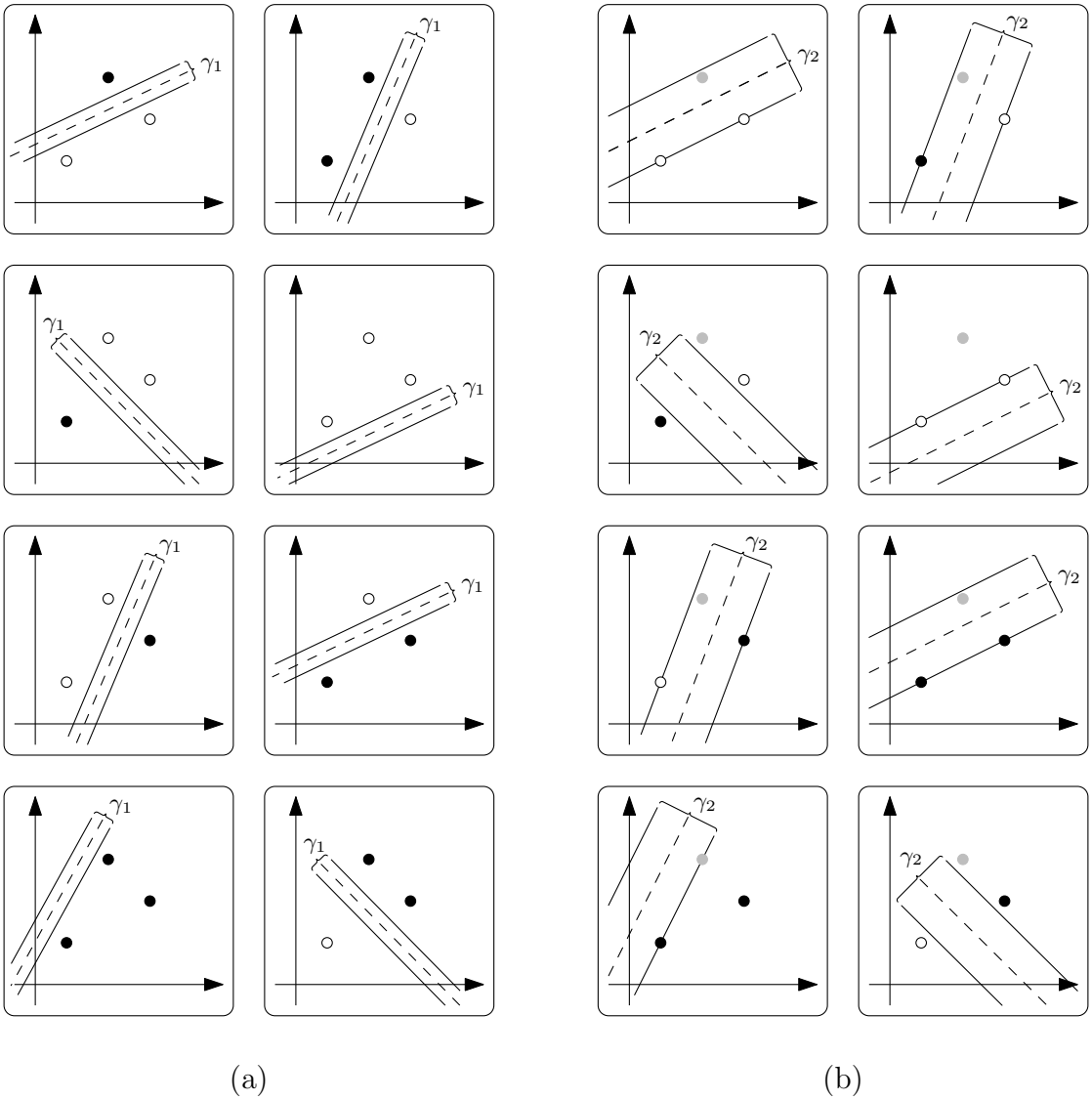



***Figure 3.7:*** *Illustration of the VC-dimension h of two classifiers $\hat{F}[\gamma_1]$ and $\hat{F}[\gamma_2]$ of decreasing complexity on an arbitrary data-set D; (a) classifier $\hat{F}[\gamma_1]$ with margin $\gamma_1$ shatters D, thus $h_{\gamma_1} = 3$, and (b) classifier $\hat{F}[\gamma_2]$ with margin $\gamma_2$ shatters only two data points, thus $h_{\gamma_2} = 2$.*

The preceding example illustrated the VC-dimension of two linear model classes, however the theoretical development shown here can be extended to include nonlinear decision surfaces for any kernel function, as well as soft-margin classifiers. It should be noted that classifier model class complexity is not only dependant on the margin, but also influenced

by the cost constant and the type of kernel used in the model class.

Using the VC-dimension $h$, the *VC-confidence* $\upsilon(k, h, \eta)$, a measure of the generalisation error of a model based upon its VC-dimension, can be calculated. It is defined as

$$\upsilon(k, h, \eta) = \sqrt{\frac{h(\log(\frac{2k}{h}) + 1) - \log(\frac{\eta}{4})}{k}}, \tag{3.48}$$

where $k$ is the size of the training data, $h$ is the VC-dimension of the classifier model, and $\eta$ is the learning rate of the classier such that $0 < \eta < 1$. Thus the upper error bound, the *generalisation error* $R[f]$ or the *generalisation bound*, is defined as

$$R[f] \leq R_{emp}[f] + \upsilon(k, h, \eta), \tag{3.49}$$

Therefore given the empirical risk and the VC-confidence of the model, one can estimate an upper bound on the expected loss of the model over the entire underlying data universe [31]. Vapnik has shown that this upper bound holds with a probability of $1 - \eta$. Figure 3.8 illustrates the relationship between the empirical risk $R_{emp}[f]$ and the VC-confidence $\upsilon(k, h, \eta)$.

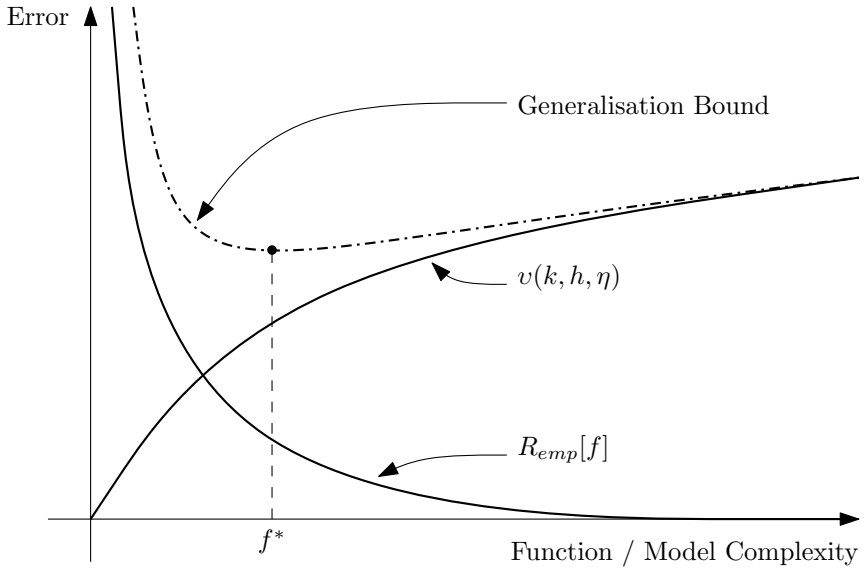



***Figure 3.8:*** *Structural Risk Minimisation.*

Minimising the generalisation bound is equivalent to making the right trade-off between model complexity and generalisation error, and thus will give the optimal classifier model $f^*$. Thus the process of Structural Risk Minimisation entails solving the optimisation problem to find the optimal classifier model $f^*$,

$$f^* = \arg\min_{f \in F} \Big( R_{emp}[f] + \upsilon(k, h, \eta) \Big), \tag{3.50}$$

where $F$ is the superclass of all model classes.

#### 3.1.1.4 SVM Training and Optimisation Techniques

Vapnik proposes a method to solve the SVM Quadratic Programming (QP) problem that has since become known as *Chunking* [40]. Vapnik's chunking algorithm involves iteratively removing rows and columns of the matrix $\boldsymbol{Q}$, each corresponding to zero-valued Lagrange multipliers, where $\boldsymbol{Q}$ is composed of elements $q_{ij} = y_i y_j \psi(\bar{\boldsymbol{x}}_i, \bar{\boldsymbol{x}}_j)$. By optimising only a small sub-set of matrix $\boldsymbol{Q}$ each iteration, chunking reduces a large QP problem into a series of smaller QP sub-problems where each iteration is initialised with the results of the previous QP sub-problem [39]. Vapnik defines an SVM as trained when the QP problem defined in Eq. 3.38, Eq. 3.39, and Eq. 3.40 is solved; an optimal feasible point is found that satisfies the KKT conditions and the matrix $\boldsymbol{Q}$ composed of elements $q_{ij} = y_i y_j \psi(\bar{\boldsymbol{x}}_i, \bar{\boldsymbol{x}}_j)$ is positive semi-definite. The KKT conditions are defined, for all $i$, as

$$\alpha_i = 0 \quad \Rightarrow \quad y_i(\bar{\boldsymbol{w}} \bullet \bar{\boldsymbol{x}}_i - b) \geq 1, \tag{3.51}$$

$$0 < \alpha_i < C \quad \Rightarrow \quad y_i(\bar{\boldsymbol{w}} \bullet \bar{\boldsymbol{x}}_i - b) = 1, \tag{3.52}$$

$$\alpha_i = C \quad \Rightarrow \quad y_i(\bar{\boldsymbol{w}} \bullet \bar{\boldsymbol{x}}_i - b) \leq 1, \tag{3.53}$$

Osuna et-al propose an extension of Vapnik's chunking technique called the QP *Decomposition Algorithm* [41]. By adding at least one example that violates the KKT conditions, thus identifying a zero-valued Lagrange multiplier, to each QP sub-problem iteration, the overall objective function maintains a feasible point that obeys all optimisation constraints, and, the sequence of QP sub-problems asymptotically converge [40]. Osuna's decomposition algorithm suggests maintaining a constant-sized matrix for each QP sub-problem, thus granting deterministic resource consumption and arbitrarily sized data sets [41].

Platt presents an algorithm for SVM training called Sequential Minimal Optimisation (SMO) that out-performs both Vapnik's chunking technique and Osuna et-al's QP decomposition algorithm [40]. Like chunking and Osuna's decomposition method, SMO decomposes the overall QP problem into smaller QP sub-problems [40]. SMO is motivated by the principle dictated by the optimisation constraint

$$\sum_{i=1}^{k} \alpha_i y_i = 0, \tag{3.54}$$

that is, $\alpha_i$ values must be simultaneously updated in pairs in order to conform to this constraint. The SMO algorithm is presented in Listing 3.2.

***Listing 3.2:*** *Sequential Minimal Optimisation algorithm*

```
1 while KKT conditions exceeded by convergence tolerance τ do
2    select α_i and α_j to update using some heuristic
3    optimise objective function φ(ᾱ) with respect to α_i and α_j while holding
         α_k(k ≠ i, j) values fixed
4 end while
```

Cichocki and Unbehauen [37] present a series of ANN-mapped techniques for the optimisation of nonlinear minimisation with mixed equality and inequality constraints. By utilising *exterior penalty function* and *interior penalty / barrier function* methods the constrained problem is approximated by an unconstrained minimisation problem. Optimisation of the unconstrained minimisation problem can then be accomplished using a standard *gradient descent* algorithm, and the derived system of differential equations can be mapped onto an ANN structure.

A constrained minimisation problem with mixed equality and inequality constraints is defined as follows. Find $\bar{\boldsymbol{x}} = [x_1, \cdots, x_n]^T \in \mathbb{R}^n$ which minimises the scalar objective function

$$f(\bar{\boldsymbol{x}}) = f(x_1, \cdots, x_n) \tag{3.55}$$

subject to the constraints

$$h_i(\bar{\boldsymbol{x}}) = 0 \qquad (i = 1, \cdots, p) \quad \text{and} \tag{3.56}$$

$$g_i(\bar{\boldsymbol{x}}) \geq 0 \qquad (i = p+1, \cdots, m). \tag{3.57}$$

According to the extended interior approach the constrained optimisation problem shown in Eq. 3.55 , Eq. 3.56, and Eq. 3.57 can be converted into an unconstrained minimisation problem by constructing an energy function of the form

$$E(\bar{\boldsymbol{x}}, \bar{\boldsymbol{\kappa}}) = f(\bar{\boldsymbol{x}}) + \sum_{i=1}^{p} \kappa_i h_i^2(\bar{\boldsymbol{x}}) + \sum_{i=p+1}^{m} \frac{1}{\kappa_i} B_i(\bar{\boldsymbol{x}}), \tag{3.58}$$

where $\kappa_i > 0$ and $B_i(\bar{\boldsymbol{x}})$ is the extended barrier function. The extended barrier function is defined as

$$B_i(\bar{\boldsymbol{x}}) = \begin{cases} \dfrac{1}{g_i(\bar{\boldsymbol{x}})}, & \text{if} \quad g_i(\bar{\boldsymbol{x}}) \geq \epsilon, \\ \dfrac{2\epsilon - g_i(\bar{\boldsymbol{x}})}{\epsilon^2}, & \text{if} \quad g_i(\bar{\boldsymbol{x}}) < \epsilon, \end{cases} \tag{3.59}$$

where $\epsilon$ is a small positive number which determines the transition from the exterior extended penalty to the interior penalty $1/g_i(\bar{\boldsymbol{x}})$.

By employing the dynamic gradient system, thus applying standard gradient descent technique, a system of differential equations can be constructed and the local minimum of the energy function shown in Eq. 3.58 can be found:

$$\frac{\mathrm{d}\bar{\boldsymbol{x}}}{\mathrm{d}t} = -\boldsymbol{\mu} \nabla_{\bar{\boldsymbol{x}}} E(\bar{\boldsymbol{x}}, \bar{\boldsymbol{\kappa}}), \tag{3.60}$$

where $\boldsymbol{\mu} = \mathrm{diag}(\mu_1, \cdots, \mu_n)$ and with the initial conditions $\bar{\boldsymbol{x}}(0) = \bar{\boldsymbol{x}}^{(0)}$. Thus

$$\frac{\mathrm{d}x_j}{\mathrm{d}t} = -\mu_j \left( \frac{\partial f(\bar{\boldsymbol{x}})}{\partial x_j} + \sum_{i=1}^{p} \kappa_i h_i(\bar{\boldsymbol{x}}) \frac{\partial h_i(\bar{\boldsymbol{x}})}{\partial x_j} + \sum_{i=p+1}^{m} \frac{1}{\kappa_i} \frac{\partial B_i(\bar{\boldsymbol{x}})}{\partial x_j} \right), x_j(0) = x_j^{(0)} \tag{3.61}$$

where $\mu_j > 0$ for all $j$ and $\kappa \geq 0$. Typically $\mu_j = \mu = 1/\tau$ for all $j$ where $\tau$ is the integration time constant, and $\kappa_i = \kappa$ for all $i$.

As the standard gradient descent algorithm will likely convergence to a local minima rather than the desired global minimum, noise can be added to the system resulting in a *stochastic gradient descent* technique [37]. By adding uncorrelated white noise to the system of differential equations,

$$\frac{\mathrm{d}\bar{\boldsymbol{x}}}{\mathrm{d}t} = -\boldsymbol{\mu}\nabla_{\bar{\boldsymbol{x}}} E(\bar{\boldsymbol{x}}, \bar{\boldsymbol{\kappa}}) + \bar{\boldsymbol{\nu}}(t), \tag{3.62}$$

where $\bar{\boldsymbol{\nu}}(t)$ is a vector of uncorrelated white noise sources with zero mean and variance decreasing in time, the process becomes stochastic and thus the likelihood of convergence to the desired global minimum is increased [37].

An ANN for convex Quadratic Programming is also presented by Cichocki and Unbehauen [37] called the *Augmented Lagrange Multiplier method*, a combination of both penalty function and ordinary Lagrange function optimisation techniques. The method is presented as follows:

$$f(\bar{\boldsymbol{x}}) = f(x_1, \cdots, x_n) \tag{3.63}$$

subject to the constraints

$$h_i(\bar{\boldsymbol{x}}) = 0 \qquad (i = 1, \cdots, p) \quad \text{and} \tag{3.64}$$

$$g_i(\bar{\boldsymbol{x}}) \geq 0 \qquad (i = p+1, \cdots, m). \tag{3.65}$$

The general augmented Lagrangian is then formed:

$$L(\bar{\boldsymbol{x}}, \bar{\boldsymbol{\lambda}}, \bar{\boldsymbol{\kappa}}) = f(\bar{\boldsymbol{x}}) + \sum_{i=1}^{p} \left( \lambda_i h_i(\bar{\boldsymbol{x}}) + \frac{\kappa_i}{2} h_i^2(\bar{\boldsymbol{x}}) \right) + \sum_{i=p+1}^{m} \left( \lambda_i g_i'(\bar{\boldsymbol{x}}) + \frac{\kappa_i}{2} g_i'^2(\bar{\boldsymbol{x}}) \right) \tag{3.66}$$

where

$$g_i'(\bar{\boldsymbol{x}}) = \begin{cases} g_i(\bar{\boldsymbol{x}}), & \text{if} \quad g_i(\bar{\boldsymbol{x}}) < -\dfrac{\lambda_i}{\kappa_i}, \\ -\dfrac{\lambda_i}{\kappa_i}, & \text{if} \quad g_i(\bar{\boldsymbol{x}}) \geq -\dfrac{\lambda_i}{\kappa_i}, \end{cases} \qquad (i = p+1, \cdots, m). \tag{3.67}$$

and $\kappa_i \geq 0$ are the penalty parameters. The minimisation of the augmented Lagrangian can be converted into a system of differential equations

$$\frac{\mathrm{d}\bar{\boldsymbol{x}}}{\mathrm{d}t} = -\boldsymbol{\mu}\nabla_{\bar{\boldsymbol{x}}} L(\bar{\boldsymbol{x}}, \bar{\boldsymbol{\lambda}}, \kappa), \tag{3.68}$$

$$\frac{\mathrm{d}\bar{\boldsymbol{\lambda}}}{\mathrm{d}t} = \quad \boldsymbol{\rho}\nabla_{\bar{\boldsymbol{\lambda}}} L(\bar{\boldsymbol{x}}, \bar{\boldsymbol{\lambda}}, \kappa), \tag{3.69}$$

where $\boldsymbol{\mu} = \mathrm{diag}(\mu_1, \cdots, \mu_n)$ and $\boldsymbol{\rho} = \mathrm{diag}(\rho_1, \cdots, \rho_m)$ are positive scalar variables, typically chosen as $\mu_i > 0$ and $\rho_i > 0$, and with the initial conditions $\bar{\boldsymbol{x}}(0) = \bar{\boldsymbol{x}}^{(0)}$ and

$\bar{\boldsymbol{\lambda}}(0) = \bar{\boldsymbol{\lambda}}^{(0)}$. Thus

$$\frac{\mathrm{d}x_j}{\mathrm{d}t} = -\mu_j \left( \frac{\partial f(\bar{\boldsymbol{x}})}{\partial x_j} + \sum_{i=1}^{p} \left( \lambda_i \frac{\partial h_i(\bar{\boldsymbol{x}})}{\partial x_j} + \frac{\kappa_i}{2} h_i(\bar{\boldsymbol{x}}) \frac{\partial h_i(\bar{\boldsymbol{x}})}{\partial x_j} \right) + \sum_{i=p+1}^{m} \left( \lambda_i \frac{\partial g_i'(\bar{\boldsymbol{x}})}{\partial x_j} + \frac{\kappa_i}{2} g_i'(\bar{\boldsymbol{x}}) \frac{\partial g_i'(\bar{\boldsymbol{x}})}{\partial x_j} \right), x_j(0) = x_j^{(0)}, \quad (3.70)$$

and

$$\frac{\mathrm{d}\lambda_j}{\mathrm{d}t} = \rho_j \left( \frac{\partial f(\bar{\boldsymbol{x}})}{\partial \lambda_j} + \sum_{i=1}^{p} \left( \lambda_i \frac{\partial h_i(\bar{\boldsymbol{x}})}{\partial \lambda_j} + \frac{\kappa_i}{2} h_i(\bar{\boldsymbol{x}}) \frac{\partial h_i(\bar{\boldsymbol{x}})}{\partial \lambda_j} \right) + \sum_{i=p+1}^{m} \left( \lambda_i \frac{\partial g_i'(\bar{\boldsymbol{x}})}{\partial \lambda_j} + \frac{\kappa_i}{2} g_i'(\bar{\boldsymbol{x}}) \frac{\partial g_i'(\bar{\boldsymbol{x}})}{\partial \lambda_j} \right), \lambda_j(0) = \lambda_j^{(0)}, \quad (3.71)$$

As this is again the standard gradient descent algorithm, convergence to a local minima, rather than the desired global minimum, is likely [37]. By adding uncorrelated white noise to the system of differential equations,

$$\frac{\mathrm{d}\bar{\boldsymbol{x}}}{\mathrm{d}t} = -\boldsymbol{\mu} \nabla_{\bar{\boldsymbol{x}}} L(\bar{\boldsymbol{x}}, \bar{\boldsymbol{\lambda}}, \kappa) + \bar{\boldsymbol{\nu}}_1(t), \quad (3.72)$$

$$\frac{\mathrm{d}\bar{\boldsymbol{\lambda}}}{\mathrm{d}t} = \boldsymbol{\rho} \nabla_{\bar{\boldsymbol{\lambda}}} L(\bar{\boldsymbol{x}}, \bar{\boldsymbol{\lambda}}, \kappa) + \bar{\boldsymbol{\nu}}_2(t), \quad (3.73)$$

where $\bar{\boldsymbol{\nu}}_1(t)$ and $\bar{\boldsymbol{\nu}}_2(t)$ are vectors of uncorrelated white noise sources with zero mean and variance decreasing in time, the process becomes stochastic and thus the likelihood of convergence to the desired global minimum is increased [37].

Larsdon et-al present another technique for solving constrained nonlinear programming problems called the *Generalised Reduced Gradient*, or GRG, algorithm [42] [43]. The Algorithm is also known as the *Conditional Gradient* or *Convex Combination* search method, and as *Frank-Wolfe* algorithm for projection-free sparse convex optimization [44]. The problem defined as

$$\min_{\bar{\boldsymbol{x}} \in \mathcal{D}} f(\bar{\boldsymbol{x}}), \quad (3.74)$$

where the objective function $f(\bar{\boldsymbol{x}})$ is convex and continuously differentiable and $\mathcal{D}$ is the feasible compact convex set of some bounded vector space, can be solved by applying the Frank-Wolfe algorithm shown in Listing 3.3.

***Listing 3.3:*** *Frank-Wolfe algorithm*

```
1  let x̄_0 ∈ D
2  for k = 0, ··· , K do
3      find s̄_k by solving arg min(s̄_k^T ∇f(x̄_k)) subject to s̄_k ∈ D
4      let γ = 2/(k+2) or find γ by solving arg min(f(x̄_k + γ(s̄_k − x̄_k))) subject to 0 ≤ γ ≤ 1
5      let x̄_{k+1} = x̄_k + γ(s̄_k − x̄_k)
6  end for
```

The Frank-Wolfe algorithm is regarded for both scalability and resultant solutions' sparse

and low-rank properties [44].

Gilbert proposes an algorithm for finding a point which is closest to the origin for some convex hull [45]. *Gilbert's algorithm* has the advantages of ease-of-implementation on general-purpose computing devices and can be analysed in a geometric manner, however, has the disadvantage of exhibiting a decreasing rate of convergence as a solution is approached, referred to as *vibration* [46]. An improved Gilbert's algorithm, with improved vibration related convergence-rate decreases, is presented by Chang et-al [46]; Chang et-al's *Improved Gilbert's Algorithm* is defined as follows. Let $\{z_i\}_{i=1}^{s_1}$ and $\{z_i\}_{i=1}^{s_2}$ be finite point sets in $\mathbb{R}^n$, $U$ and $V$ denote two convex polytopes, where

$$U = \{u = \sum_{i=1}^{s_1} \alpha_i z_i \quad | \quad \alpha_i \geq 0, \quad \sum_{i=1}^{s_1} \alpha_i = 1\}, \tag{3.75}$$

and

$$V = \{v = \sum_{i=1}^{s_2} \alpha_i z_i \quad | \quad \alpha_i \geq 0, \quad \sum_{i=1}^{s_2} \alpha_i = 1\}. \tag{3.76}$$

The aim of a general nearest point problem is to find the nearest points between the two polytopes,

$$\min_{u \in U, v \in V} \|u - v\|. \tag{3.77}$$

Gilbert's algorithm transforms the nearest point problem into one which finds the minimal distance from the origin to a polytope. The Minkowski set difference of $U$ and $V$ is denoted $Z = U - V$, thus $Z$ is a polytope which has $s_1 \times s_2$ vertices. It follows that the nearest point problem is now the minimum distance problem:

$$\min_{z \in Z} \|z\|. \tag{3.78}$$

The mathematical notation of the support properties of a convex polytope are presented below. Suppose $P$ is a convex polytope and $P = \mathcal{C}\{Z\}$. The support function $h_P : \mathbb{R}^n \to \mathbb{R}$ is defined as

$$h_z(\eta) = \max_{z \in Z} \eta \cdot z. \tag{3.79}$$

The solution to $h_z(\eta)$ is defined as $s_Z(\eta)$, where $s_Z(\eta)$ satisfies

$$h_Z(\eta) = s_Z(\eta) \cdot z \quad \text{and} \quad s_Z(\eta) \in Z. \tag{3.80}$$

The function $g_P : \mathbb{R}^n \times P \to \mathbb{R}$ is defined as

$$g_P(\eta, p) = h_Z(\eta) - \eta \cdot p. \tag{3.81}$$

When $z \in Z$, then $z$ is the solution of the minimum distance problem, shown in Eq. 3.78, if and only if $g_Z(-z, z) = 0$.

The vibration phenomena leads to slow convergence to the aforementioned solution to

the minimum distance problem. In the periodic vibration case, periodically occurring vertices are chosen to calculate the solution to the minimum distance problem. Denoting $P_1, \cdots, P_n$ as the periodically occurring vertices of the convex polytope, $\mathcal{C}\{P\}$ is the convex hull, $z_1, \cdots, z_i$ is the iterative point set calculated in the algorithm, as shown in Listing 3.4, and $z^*$ is the point set's limit-point and the solution to the minimum distance problem.

Chang et-al have shown that if the algorithm vibrates among the vertices $P_1, \cdots, P_n$, then the limit point $z^*$ is in the convex hull $\mathcal{C}\{P_1, \cdots, P_n\}$, and all vibration points belong to a supporting hyperplane [46]. The *Improved Gilbert's algorithm* calculates the minimum distance from origin to the convex combination of vibration points, where the improved nearest point algorithm defined as

$$z^* : \|z^*\| = \min_{z \in \mathcal{C}\{P_1, \cdots, P_n\}} \|z\|. \tag{3.82}$$

In terms of SVM training, the number of support vectors are usually always much less than the number of training examples, and the optimisation shown in Eq. 3.82 can be solved by

$$\min_{\bar{\boldsymbol{\lambda}}} \|\lambda_1 P_1 + \cdots + \lambda_n P_n\|^2, \tag{3.83}$$

subject to the constraints

$$0 \leq \lambda_i \leq 1, \quad \text{and} \tag{3.84}$$

$$\lambda_i + \cdots + \lambda_n = 1, \tag{3.85}$$

where $i = 1, \cdots, n$. The problem shown in Eq. 3.83 can be solved by the Lagrange method and transforming it into a set of $2n + 1$ linear equations; solving by linear methods results in $\bar{\boldsymbol{\lambda}} = \{\lambda_1^*, \cdots, \lambda_n^*\}$ and the nearest point can be calculated by $z^* = \lambda_1^* P_1 + \cdots + \lambda_n^* P_n$.

The Improved Gilbert's algorithm is shown in Listing 3.4.

***Listing 3.4:*** *Improved Gilbert's algorithm*

```
choose initial value z_0 ∈ Z
for i ≤ k_0, where i is iteration count and k_0 is the iteration-count ceiling, do
    compute h_Z(−z_i) and z̄_i = s_Z(−z_i)
    compute g_Z(−z_i, z_i)
    if g_Z(−z_i, z_i) = 0
        z_i = z*
        end algorithm
    find nearest point along line segment z_{i+1} = z_i z̄_i
    increment iteration count i
end for
get periodically repeated vertices A = {P_1, ··· , P_l}
calculate the nearest point z* that lies on the hyperplane defined by
    A = {P_1, ··· , P_l}
```

#### 3.1.1.5 Multi-class SVM Classifiers

Real-world problems often require the classification of objects into more than two classes. This section will introduce several multi-class classification approaches utilising binary SVMs.

*One-versus-the-rest classification* is the most popular technique for multi-class classification using binary support vector machines [31]. Consider the training set

$$D = \{(\bar{\boldsymbol{x}}_1, y_1), (\bar{\boldsymbol{x}}_2, y_2), \cdots, (\bar{\boldsymbol{x}}_k, y_k)\} \subseteq \mathbb{R}^n \times \{1, 2, \cdots, M\}, \tag{3.86}$$

where the label $y_i$ for each observation can take on any value in $\{1, 2, \cdots, M\}$ with $M > 2$, and $\{1, 2, \cdots, M\}$ is the unique label set. In the one-versus-the-rest technique, given $M$ unique classes, $M$ binary support vector-based decision surfaces, $g^1, \cdots, g^M$, are constructed. Each decision surface is trained to separate one class from the rest [31]. To classify unknown points, a voting scheme is used based on which of the $M$ decision surfaces returns the largest value, and thus assign a class label accordingly.

To train $M$ decision surfaces $M$ training sets are constructed,

$$D^p = D^p_+ \cup D^p_-, \tag{3.87}$$

where

$$D^p_+ = \{(\bar{\boldsymbol{x}}, +1) \mid (\bar{\boldsymbol{x}}, y) \in D \wedge y = p\}, \tag{3.88}$$

and

$$D^p_- = \{(\bar{\boldsymbol{x}}, -1) \mid (\bar{\boldsymbol{x}}, y) \in D \wedge y \neq p\}, \tag{3.89}$$

where $p = 1, \cdots, M$. The set $D^p_+$ contains all the observations in $D$ that are members of the class $p$, and the set $D^p_-$ contains all the remaining observations. Each decision surface $g^p$ is then trained on the corresponding dataset $D^p$ which gives rise to the surface of the form

$$g^p(\bar{\boldsymbol{x}}) = \sum_{i=1}^{|D^p|} \alpha_i^p y_i \psi(\bar{\boldsymbol{x}}_i, \bar{\boldsymbol{x}}) - b^p, \tag{3.90}$$

where $\bar{\boldsymbol{x}}_i, y_i) \in D^p$. The same cost constant and kernel function are used for the training of the $M$ decision surfaces. Thus a decision function can be constructed

$$\hat{f}(\bar{\boldsymbol{x}}) = \arg\max_p g^p(\bar{\boldsymbol{x}}), \tag{3.91}$$

where $p \in \{1, 2, \cdots, M\}$. The decision function returns the label of the decision surface that assigns some point $\bar{\boldsymbol{x}} \in \mathbb{R}^n$ to its $+1$ class with the highest confidence [31].

Although the one-versus-the-rest classification technique has shown to be robust in real-world applications, the unbalanced nature of training sets can lead to potential misclas-

sification of points [31]. The *pairwise classification* technique avoids this situation by constructing decision surfaces for each pair of classes, and again, the classification of an unknown point is achieved by a voting scheme.Consider the training set

$$D = \{(\bar{\boldsymbol{x}}_1, y_1), (\bar{\boldsymbol{x}}_2, y_2), \cdots, (\bar{\boldsymbol{x}}_k, y_k)\} \subseteq \mathbb{R}^n \times \{1, 2, \cdots, M\}, \tag{3.92}$$

in pairwise classification $M(M-1)/2$ decision surfaces are constructed; one decision surface for each possible pair of classes. Let $g^{p,q} : \mathbb{R}^n \to \{p, q\}$ denote the decision surface that separates the pair of classes $p$ and $q$ with $p = q$ and $\{p, q\} \subset \{1, 2, \cdots, M\}$. The decision surface $g^{p,q}(\bar{\boldsymbol{x}})$ is trained,

$$g^{p,q}(\bar{\boldsymbol{x}}) = \sum_{i=1}^{|D^{p,q}|} \alpha_i^{p,q} y_i \psi(\bar{\boldsymbol{x}}_i, \bar{\boldsymbol{x}}) - b^{p,q}, \tag{3.93}$$

on the dataset

$$D^{p,q} = D^p \cup D^q, \tag{3.94}$$

where

$$D^p = \{(\bar{\boldsymbol{x}}, y) \mid (\bar{\boldsymbol{x}}, y) \in D \wedge y = p\}, \tag{3.95}$$

and

$$D^q = \{(\bar{\boldsymbol{x}}, y) \mid (\bar{\boldsymbol{x}}, y) \in D \wedge y = q\}. \tag{3.96}$$

The set $D^p$ consists of all the observations in $D$ with the label $p$ and the set $D^q$ consists of al the observations in $D$ with the label $q$. The training set $D^{p,q}$ for the pair of classes $p$ and $q$ is simply the union of these two sets [31]. To classify an unknown point each of the $M(M-1)/2$ decision surfaces are applied to the point, keeping track of how many times the point was assigned to what class label. The class label with the highest count is then considered the label for the unknown point.

Two other methods are frequently mentioned in the literature: *error-correcting-output-codes classification*, and *multi-objective support vector machine*. Both of these approaches have nice theoretical properties but are not often used in practise due to computation complexity [31]; thus these approaches will not be discussed further here.

#### 3.1.1.6 Regression and Prediction with SVMs

In regression problems observations are associated with a numerical value rather than a label from a set of discrete labels. Thus the definition of machine learning can be adapted to deal explicitly with numerical training observations [31]. The following is the adapted definition of machine learning for regression. Given a data universe $X$, a sample set $S$ where $S \subset X$, some target function $f : X \to \mathbb{R}$, and a training set $D$ where $D = \{(x, y) \mid x \in S \text{ and } y = f(x)\}$, compute a model $\hat{f} : X \to \mathbb{R}$ using $D$ such that

$$\hat{f}(x) \cong f(x), \tag{3.97}$$

for all $x \in X$. From a statistical perspective, *linear regression* is the fitting of a hyperplane through a set of training points with a minimum error. The regression error is characterised by *residual terms*, which are defined as the difference between the output of the model and the actual value of the training observations. The goal in linear regression is to minimise these residuals [31]. Assume a regression training set of the form

$$D = \{(\bar{\boldsymbol{x}}_1, y_1), (\bar{\boldsymbol{x}}_2, y_2), \cdots, (\bar{\boldsymbol{x}}_k, y_k)\} \subset \mathbb{R}^n \times \mathbb{R}. \tag{3.98}$$

Assume that $\hat{f}(\bar{\boldsymbol{x}})$ is a regression model on $D$; then the quantity

$$\rho_i = y_i - \hat{f}(\bar{\boldsymbol{x}}_i) \tag{3.99}$$

for $(\bar{\boldsymbol{x}}_i, y_i) \in D$, called a *residual*, measures the difference between model output and the actual observation. For a perfect model the residuals are all zero, however in real-word situations this is unlikely to occur. Thus one must construct models where the residuals are minimised [31]. In linear regression this is accomplished by computing the minimum *sum of squared errors*,

$$\min \sum_{i=1}^{k} \rho_i^2 = \min_{\hat{f}} \sum_{i=1}^{k} \Big( y_i - \hat{f}(\bar{\boldsymbol{x}}_i) \Big)^2, \tag{3.100}$$

where $(\bar{\boldsymbol{x}}_i, y_i) \in D$. Therefore the optimisation problem that computes the optimal linear regression model $\hat{f}^*$ is given as

$$\hat{f}^* = \arg\min_{\hat{f}} \sum_{i=1}^{k} \Big( y_i - \hat{f}(\bar{\boldsymbol{x}}_i) \Big)^2. \tag{3.101}$$

As the regression models are linear, Equation 3.101 can be rewritten as

$$(\bar{\boldsymbol{w}}^*, b^*) = \arg\min_{\bar{\boldsymbol{w}}, b} \sum_{i=1}^{k} \Big( y_i - \bar{\boldsymbol{w}} \bullet \bar{\boldsymbol{x}}_i + b \Big)^2, \tag{3.102}$$

where the optimal regression model is

$$\hat{f}^*(\bar{\boldsymbol{x}}) = \bar{\boldsymbol{w}}^* \bullet \bar{\boldsymbol{w}} - b^*. \tag{3.103}$$

The development of the support vector regression machine is very similar to the development of support vector machine for classification [31]. Recall that the SVM classification problem utilises underpinnings of a maximum-margin classifier - a hyperplane is found based on the distances of the observations to that hyperplane. For the SVM regression problem a hypertube of width $2\epsilon$, $\epsilon > 0$, and with a hyperplane positioned right in its center is found that accurately models all of the observations. Thus given the regression training set

$$D = \{(\bar{\boldsymbol{x}}_1, y_1), (\bar{\boldsymbol{x}}_2, y_2), \cdots, (\bar{\boldsymbol{x}}_k, y_k)\} \subseteq \mathbb{R}^n \times \mathbb{R}, \tag{3.104}$$

one can compute the optimal linear support vector regression model $\hat{f}^*(\bar{\boldsymbol{x}}) = \bar{\boldsymbol{w}}^* \bullet \bar{\boldsymbol{w}} - b^*$ with the dual optimisation problem

$$\max_{\bar{\boldsymbol{\alpha}}^\bullet, \bar{\boldsymbol{\alpha}}^\circ} \phi'(\bar{\boldsymbol{\alpha}}^\bullet, \bar{\boldsymbol{\alpha}}^\circ) = \max_{\bar{\boldsymbol{\alpha}}^\bullet, \bar{\boldsymbol{\alpha}}^\circ} \left( -\frac{1}{2} \sum_{i=1}^{k} \sum_{j=1}^{k} (\alpha_i^\bullet - \alpha_i^\circ)(\alpha_j^\bullet - \alpha_j^\circ) \bar{\boldsymbol{x}}_i \bullet \bar{\boldsymbol{x}}_j + \sum_{i=1}^{k} y_i(\alpha_i^\bullet - \alpha_i^\circ) - \epsilon \sum_{i=1}^{k} (\alpha_i^\bullet + \alpha_i^\circ) \right), \quad (3.105)$$

subject to the constraints

$$\sum_{i=1}^{k} (\alpha_i^\bullet - \alpha_i^\circ) = 0, \quad (3.106)$$

$$C \geq \alpha_i^\bullet, \text{ and} \quad (3.107)$$

$$\alpha_i^\circ \geq 0, \quad (3.108)$$

for $i = 1, \cdots, k$, where

$$\bar{\boldsymbol{w}}^* = \sum_{i=1}^{k} (\alpha_i^\bullet - \alpha_i^\circ)^* \bar{\boldsymbol{x}}_i, \quad (3.109)$$

$$b^* = \frac{1}{k} \sum_{i=1}^{k} \bar{\boldsymbol{w}}^* \bullet \bar{\boldsymbol{x}}_i - y_i. \quad (3.110)$$

In support vector regression models one can interpret an observation $(\bar{\boldsymbol{x}}_i, y_i)$ for which the coefficient $(\alpha_i^\bullet - \alpha_i^\circ)$ is non-zero as a support vector [31]. Therefore the optimal model for a linear support vector regression machine is given by

$$\begin{aligned} \hat{f}^*(\bar{\boldsymbol{x}}) &= \bar{\boldsymbol{w}}^* \bullet \bar{\boldsymbol{w}} - b^* \\ &= \sum_{i=1}^{k} (\alpha_i^\bullet - \alpha_i^\circ)^* \bar{\boldsymbol{x}}_i \bullet \bar{\boldsymbol{x}} - \frac{1}{k} \sum_{i=1}^{k} \sum_{j=1}^{k} (\alpha_i^\bullet - \alpha_i^\circ)^* \bar{\boldsymbol{x}}_i \bullet \bar{\boldsymbol{x}}_j - y_j. \end{aligned} \quad (3.111)$$

One can extend the linear support vector regression machine to a nonlinear support vector regression machine using the kernel trick [31]. Thus one can compute the optimal nonlinear support vector regression model $\hat{f}^*(\bar{\boldsymbol{x}})$ with the dual optimisation problem

$$\max_{\bar{\boldsymbol{\alpha}}^\bullet, \bar{\boldsymbol{\alpha}}^\circ} \phi'(\bar{\boldsymbol{\alpha}}^\bullet, \bar{\boldsymbol{\alpha}}^\circ) = \max_{\bar{\boldsymbol{\alpha}}^\bullet, \bar{\boldsymbol{\alpha}}^\circ} \left( -\frac{1}{2} \sum_{i=1}^{k} \sum_{j=1}^{k} (\alpha_i^\bullet - \alpha_i^\circ)(\alpha_j^\bullet - \alpha_j^\circ) \psi(\bar{\boldsymbol{x}}_i, \bar{\boldsymbol{x}}_j) + \sum_{i=1}^{k} y_i(\alpha_i^\bullet - \alpha_i^\circ) - \epsilon \sum_{i=1}^{k} (\alpha_i^\bullet + \alpha_i^\circ) \right), \quad (3.112)$$

subject to the constraints

$$\sum_{i=1}^{k}(\alpha_i^{\bullet} - \alpha_i^{\circ}) = 0, \tag{3.113}$$

$$C \geq \alpha_i^{\bullet}, \text{ and} \tag{3.114}$$

$$\alpha_i^{\circ} \geq 0, \tag{3.115}$$

for $i = 1, \cdots, k$. The optimal model for a nonlinear support vector regression machine is given by

$$\hat{f}^*(\bar{\boldsymbol{x}}) = \sum_{i=1}^{k}(\alpha_i^{\bullet} - \alpha_i^{\circ})^*\psi(\bar{\boldsymbol{x}}_i, \bar{\boldsymbol{x}}) - \frac{1}{k}\sum_{i=1}^{k}\sum_{j=1}^{k}(\alpha_i^{\bullet} - \alpha_i^{\circ})^*\psi(\bar{\boldsymbol{x}}_i, \bar{\boldsymbol{x}}_j) - y_j. \tag{3.116}$$

### 3.1.2 Unsupervised Learning

There exist two fundamental machine learning paradigms; *Learning with a Teacher*, or *Supervised Learning*, and the logical corollary, *Learning without a Teacher*, *Unsupervised Learning*, or *Self-organising Learning*. Learning with a teacher has already been presented in Section 3.1.1.3, where the learning process was directed by a supervisor that supplied labels for a set of input vectors according to some unknown but fixed probability density function. Learning without a teacher, as the nomenclature suggests, that there is no supervisor to supply labels for the set of input vectors to the learning machine [13]. Rather, provision is made for a *task-independent measure* of the quality of representation that the machine is required to learn, and the free parameters of the machine are optimised with respect to that measure using a competitive learning rule [13].

#### 3.1.2.1 Legacy SVM System

Rajkumar has designed an oil and gas pipeline defect-monitoring and failure-prediction system utilising *k-means clustering* and *SVM classification and regression* subsystems [29]. Input data to the system is generated by measuring reflected pulses directed down the length of the pipeline by ultrasonic transducers. Processing of the input data by the underlying $k$-means clustering and SVM subsystems can lead to the identification of present defects, and, predict time to pipeline failure. Due to the computationally demanding nature of the underlying mathematics of the SVM subsystems Rajkumar's prototypes do not achieve real-time performance. Thus an enhanced real-time implementation of these subsystems is required for the development and deployment of the pipeline defect-monitoring and failure-prediction system in the oil and gas industries.

Figure 3.9 illustrates the machine learning subsystems of Rajkumar's oil and gas pipeline defect-monitoring and failure-prediction system [29].

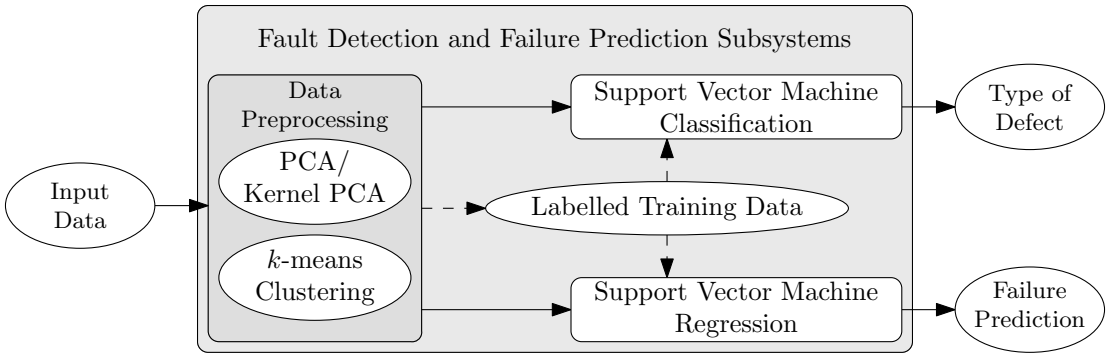



***Figure 3.9:*** *System-level diagram of Rajkumar's oil and gas pipeline defect-monitoring and failure-prediction subsystems.*

#### 3.1.2.2 k-Means Clustering

*k-Means Clustering* [47], [48] is a method of partitioning a set of data points into some fixed number of clusters based on each data point's mean.

Given a data set $D = \{\bar{\boldsymbol{x}}_1, \bar{\boldsymbol{x}}_2, \cdots, \bar{\boldsymbol{x}}_k\} \subseteq \mathbb{R}^n$, k-means clustering aims to partition the $k$ data points into $n$ sets $S = \{\boldsymbol{S}_1, \boldsymbol{S}_2, \cdots, \boldsymbol{S}_n\}$, where $n \leq k$, so as to minimize the square of the distance from the data points to their assigned cluster's mean or centre point $\bar{\boldsymbol{\mu}}_i$. Thus k-means clustering optimises

$$\min_S \sum_{i=1}^{n} \sum_{\bar{\boldsymbol{x}} \in \boldsymbol{S}_i} \|\bar{\boldsymbol{x}} - \bar{\boldsymbol{\mu}}_i\|^2. \tag{3.117}$$

### 3.1.3 SVM Hardware Implementations

Recent research on FPGA based SVMs with supervised training have shown varying results [49], [50], [51], [52]. Anguita *et al.* [49], [50] have implemented and investigated a fixed-point number representation realisation of the SVM training subsystem, a method of solving the QP problem. Their system solves the QP problem using a simple ANN based bisection method to find $b^*$, however, with unsatisfactory performance in terms of operation time due in part to the technology available at the time of publication and the complexity and feed-forward nature of the implemented hardware [50]. No mention is made of whether real-time performance is achieved. The performance in terms of errors due to quantisation noise was however encouraging [49]. Kim *et al.* has implemented a real-time FPGA SVM system [51]. Exploiting a parallel architecture with a two-stage pipeline the system is accelerated to process large amounts of data for real-time classification [51].

Irick *et al.* has proposed an FPGA architecture that operates in signed logarithm number representation system, where multiply operations are replaced with addition, and all mathematical operations are by means of look-up tables [53]. This implementation however lacks a training subsystem; support vectors are fed directly to the SVM classification

subsystem. Ruiz-Llata *et al.* has presented an SVM architecture for classification and regression, however as with Irick's design, support vectors and training must be trained externally to the system, and provided at run time [54].

Gomes Filho *et al.* has presented a hardware implementation of the sequential minimal optimisation training phase for SVM using floating-point number representation and partial reconfiguration FPGA technology [55]. The system performed well enough, though no mention of real-time performance was made, however by utilising partial reconfiguration technology the FPGA area was reduced by over 20% against an acceptable reconfiguration penalty time [55]. Patil *et al.* has also presented a partial reconfiguration based FPGA-based SVM architecture [56]. The design also make use of systolic array architecture to provide efficient memory management, reduced complexity, and efficient data transfer mechanisms [56]. The systolic array architecture is partially reconfigured as required, and has been shown to reduce power consumption during the classification phase [56].

The QP problem has been identified as the computationally demanding component of all the systems reviewed in the literature [57]. In the hardware domain three different approaches to QP solving have been investigated with clear conclusive results; one method's architecture conclusively out-performs the others. Gilbert's algorithm [45], [46] has been actualised within an FPGA-based SVM training subsystem and has shown to accelerate the performance compared with implementations of the nearest-point algorithm and sequential minimal optimisation [58], [57], [59], [60], [61], [62], [63], [64], [65].

Gilbert's algorithm solves the QP problem from a geometric perspective. By iterating through the algorithm, the QP problem space, regarded as a *convex hull*, finds the minimum point on the convex hull geometric object, thus solving the optimisation problem [45], [46].

Afifi *et al.* present an FPGA-accelerated SVM classifier co-processor for classification of melonoma skin-cancer images [2], [3]. Afifi *et al.* reaffirm the computationally intensive task of SVM classification is a suitable candidate for hardware acceleration through the use of an FPGA-based SVM classification subsystem co-processor. The hardware and software co-design was implemented using the Xilinx Zynq 7000 device, a System on a Chip (SoC) platform combining Xilinx FPGA fabric with an ARM Cortex-A9 dual-core processor, and exploiting Xilinx Vivado Design Suite's High-Level Synthesis (HLS) design methodology and tool allowing high-level C/C++ software languages to be used in place of tradition Hardware Description Language (HDL) FPGA design and implementation strategies. The SVM classification co-processor was implemented using the SVM-Light project's C source code and the Xilinx HLS proprietary Intellectual Property (IP) to accelerate and simplify the design and implementation of the system.

Afifi *et al.* SVM co-processor implementation on the Xilinx Zync 7000 platform demonstrated high performance with low resource utilisation and power consumption thus meeting constraints for deployment as an embedded system [2], [3]. Table 3.2 provides a summary of Afifi *et al.* Zync 7000 FPGA-based SVM Classifier co-processor Device Utilisation. Table 3.3 provides a summary of Afifi *et al.* Zync 7000 FPGA-based SVM Classifier co-processor On-Chip Components Power Consumption.

***Table 3.2:*** *Afifi* et al. *Zync 7000 FPGA-based SVM Classifier co-processor Device Utilisation Summary*

| Resource | Utilisation % |
|---|---|
| Slice FF Registers | 5.25 |
| Slice LUTs | 8.22 |
| Memory LUT | 0.99 |
| BRAM | 2.14 |
| DSP48 | 2.27 |
| BUFG | 3.13 |

***Table 3.3:*** *Afifi* et al. *Zync 7000 FPGA-based SVM Classifier co-processor On-Chip Components Power Consumption Summary*

| On-Chip Component | Power (mW) |
|---|---|
| Clocks | 9 |
| Logic | 3 |
| Signals | 4 |
| BRAM | 2 |
| DSPs | <1 |
| PS7 | 1565 |
| Total Dynamic Power | 1584 |
| Device Static Power | 154 |
| Total On-Chip Power | 1738 |

## 3.2 Digital Logic and Field Programmable Gate Arrays

Field Programmable Gate Arrays (FPGAs) are digital integrated circuits (ICs) that contain configurable blocks of logic along with configurable interconnects between these blocks [66]. A modern approach to very large and complex FPGA design and implementation involves describing one's system by means of a HDL such as VHDL and Verilog HDL [67]. The work-flow is similar to that of the design and implementation of a Finite State Machine (FSM) with discrete digital logic, however, from the state diagram representation stage follows the coding of a circuit description in HDL [67], [68]. Synthesis and circuit instantiation is usually performed by some proprietary tool supplied and licensed from the FPGA hardware vendor.

### 3.2.1 Field Programmable Gate Array Logic

A complete FPGA comprises a large number of *Programmable Logic Blocks* surrounded by a sea of *Programmable Interconnects* (PIs), similar to the architecture found in CPLD devices, as shown in Fig. 3.10 [66]; this is of course an abstract illustration of the structures formed by transistors and interconnects on the same piece of silicon semiconductor. It is the configuration of each PI that dictates how the PLBs within the sea of PIs are connected .

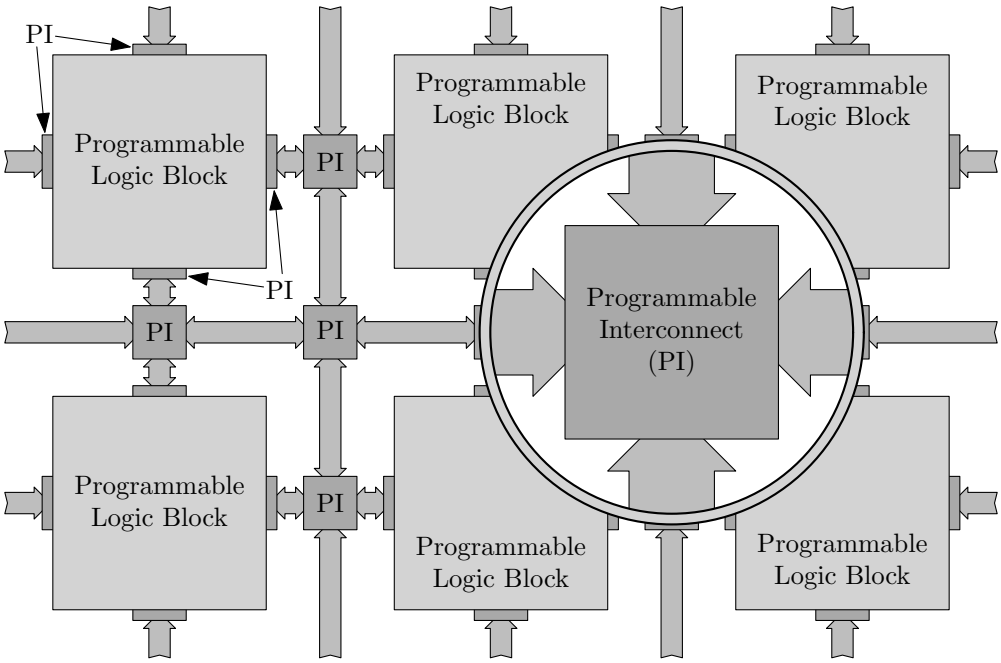



***Figure 3.10:*** *Illustration of generic FPGA architecture, also referred to as fabric, with generic terminology, as viewed from above.*

Altera devices have a Programmable Logic Block structure known as a *Logic Array Block* (LAB) [68]. LABs Figure 3.11 illustrates Altera's FPGA digital logic structure and sub-structure architecture, referred to as the *fabric*. Each LAB comprises four *Adaptive Logic Modules* (ALMs), of which each contain two *Logic Elements* (LEs).

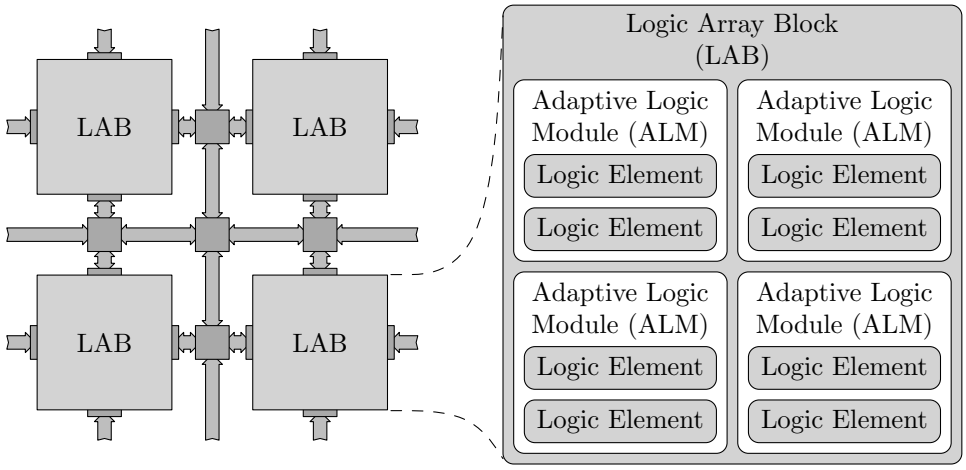



***Figure 3.11:*** *Illustration of Altera FPGA architecture or fabric as viewed from above.*

A generalisation of both Altera Logic Element (LE) FPGA sub-structure architectures as a quantum unit is shown in Fig. 3.12 [68]. The LE comprises an $n$-input LUT that can serve as $n^2$-bit RAM or an $n^2$-bit *Shift Register* (SR), a *multiplexor* (MUX), and a single-bit register shown as a D Flip-Flop in Fig. 3.12 [66]. The register can be configured to act as a flip-flop or as a latch. The Clock edge-triggering polarity, Clear or Clock-Enable and Set / Reset or Preset signal polarity can also be configured. There are also other digital-logic elements not shown in Fig. 3.12; these elements include fast carry logic for use in arithmetic operations [68]. The exact architecture of these sub-structures vary from manufacturer to manufacturer, across FPGA device families, and within individual family revision and generation [66]. It is within these LEs that an FPGA's custom digital logic is configured, and is subsequently connected accordingly within the whole FPGA device's fabric by each LAB's adjacent PIs.

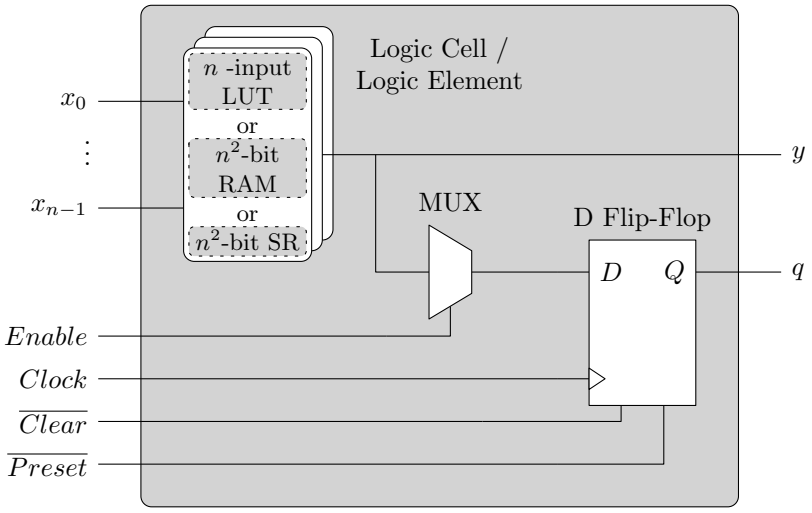



***Figure 3.12:*** *Illustration of the generalised Altera Logic Element FPGA architectures as a quantum unit.*

Many applications require the use of significant portions memory; FPGAs now include relatively large blocks of embedded Random Access Memory (RAM) often arranged in columns amongst a device's fabric [68]. In addition to memory, many applications require the use of large numbers of arithmetic function logic. These functions, namely multiplier and multiply-and-accumulate, are inherently slow if they are implemented by connecting large numbers of programmable logic blocks together, thus fast hard-wired arithmetic function blocks are incorporated in FPGA device architectures [66], [68]. A generic FPGA architecture with embedded RAM and multiplier or Multiply-Accumulate (MAC) function blocks set in columns amongst the programmable logic block fabric of the device is shown in Fig. 3.13.

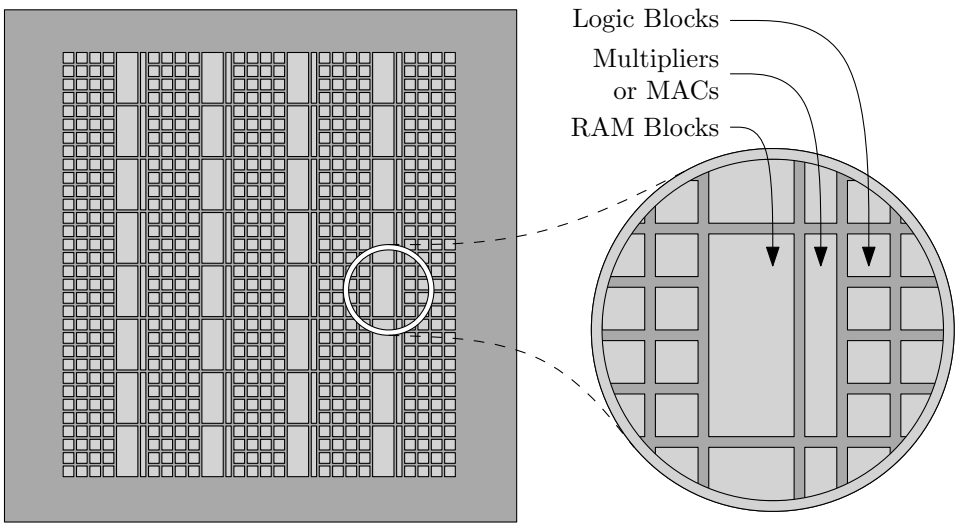



***Figure 3.13:*** *A generic FPGA architecture with embedded RAM and multiplier or MAC instruction blocks arranged in columns amongst the programmable logic block fabric of the device.*

Any digital electronics can be realised in either hardware, built from logic gates and various memory registers, or in software, where individual instructions are executed sequentially on a single microprocessor or on parallel microprocessors. The speed that one wishes their design to operate will usually determine whether a design is implemented purely in hardware or software, or as a combination of both [66].

### 3.2.2 FPGAs and the Integrated Circuit Market

FPGAs are currently consuming the market-share of all other digital IC technologies. The technologies falling victim to the rise of the FPGA include ASIC and *custom silicon*, *Digital Signal Processing technologies* (DSP) , *embedded microcontroller and microprocessor devices*, and *physical-layer communications devices*. FPGA technology has also spawned a new growing market of its own, known as *Reconfigurable Computing* (RC) [66].

The trend has been to include a collection of heterogeneous complex hardware units, such as DSP blocks, high-speed communications blocks, and both hard and soft microprocessors, embedded within FPGA technology [68]. This has heralded the evolution of FPGA designs that not only implement digital logic through the utilisation of LUTs and various registers and memories at a low-level circuit architecture scale, but also the inclusion of the embedded functional blocks that create higher-level system designs, all in the one FPGA IC; hence the emergence of the SoC / VLSI paradigm [68]. Tools are also evolving from the hardware vendors that enable a more timely design and implementation period, by utilising these functional blocks and vendor specific IP [66] [69]. These factors continue to ensure FPGA technology market growth and the wide-scale adoption, use, and technological evolution [66], [68].

## 3.3 Digital Signal Processing

Digital Signal Processing (DSP) is used in a wide range of applications, including high-definition TV, mobile telephony, digital audio, multimedia, digital cameras, radar, sonar detectors, biomedical imaging, global positioning, digital radio, speech recognition, to name but a few [68]. The field has been driven by the ever-increasingly demanding application requirements, and has been supported from an evolving integrated circuit (IC) industry rife with the developments programmable digital logic technologies. The development of programmable DSP ICs and dedicated System-on-a-Chip (SoC) solutions for these applications has been an active area of research and development over the last three decades, and has spawned a class of dedicated microprocessors, known as DSP microprocessors, or just DSPs, that specifically target many of these application [68]. Also, with the introduction of FPGAs, DSP engineers have never had such a vast, accessible, and competent set of tools and technologies at their finger-tips to apply to the ever-increasing list of possible DSP applications.

### 3.3.1 Practical DSP Fundamentals

The choice of algorithm and arithmetic requirements in a DSP system can have severe implications on the quality of the final implementation [68]. Some of these issues and considerations that have not been covered in Chapter 2 will be presented in this section.

*Latency* is the time $t_p$ required to produce a result after an input DATA is fed into the system, as shown in Fig. 3.14. In synchronous systems this is defined as the number of clock cycles which must evolve before the output is produced [68]. *Throughput* is the time between each successive output. Figure 3.14 shows a system whose throughput is limited by the `dsp_en` signal asserting `adc_data` DATA at a period of $t_p + t_n$. *Pipelining* stages of the DSP circuit will have the positive effect of increased throughput, but at the cost of increased latency, circuit size, and power requirements.

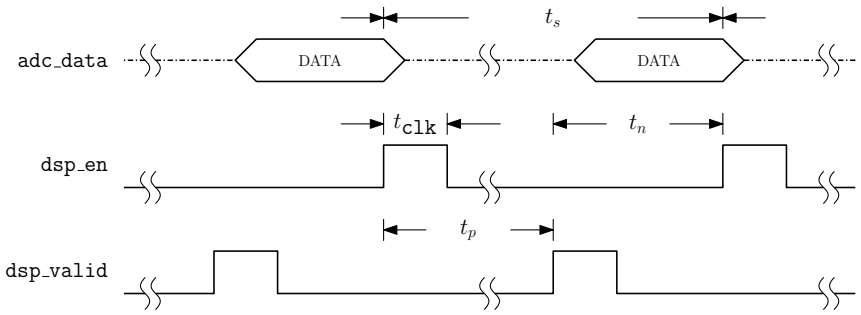



***Figure 3.14:*** *DSP processing latency.*

### 3.3.2 Parallel Machines and Systolic Signal Processing

Given the scope of this research endeavour, special mention must be made of parallelism of DSP hardware, known as *Systolic Signal Processing*. While the sequential model, as seen in single-core microprocessor based DSP systems, is capable of implementing a wide range of algorithms, significant performance gains are observed in parallel-processing DSP systems implementations [68]. Systolic array architectures were first introduced into VLSI design by Kung and Leiserson in 1978 [68], [70]. Systolic array architectures have the following features; an array of processors with extensive concurrency, a small number of processor types, array control of is simple, and interconnection within the array are local [71], [72], [73].

Systolic arrays processing power comes from the concurrent use of many simple cells, as opposed to the sequential use of very powerful cells, and are particularly suitable for parallel algorithms with simple and regular dataflows, such as matrix and vector based operations [68], [72]. By pipelining operations within a systolic array, further processing power can be observed through the efficient use of all processing cells. Figure 3.15, Fig. 3.16, and Fig. 3.17 illustrate various systolic array architecture structures; the black circles represent pipeline stages after each *processing element* (PE), and the lines drawn through the pipeline stages are the scheduling lines that depict which PEs are operating on the same iteration at the same time, the calculations performed at the same clock-cycle [68].

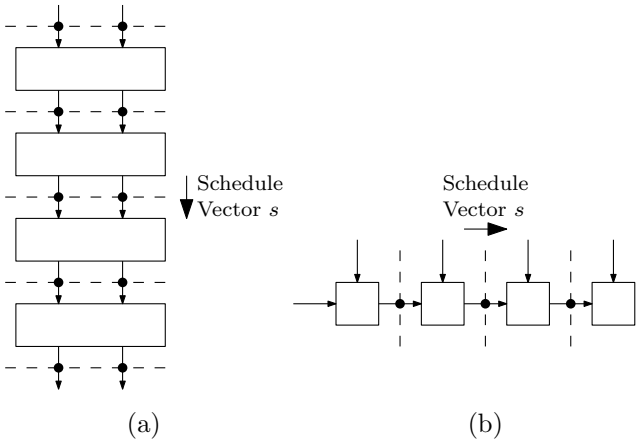



***Figure 3.15:*** *Linear systolic array architectures; (a) column, and (b) row.*

Rectangular systolic arrays, as shown in Fig. 3.16(a), are highly suitable for matrix operations [68]. In all the systolic arrays illustrated each PE receives data only from its nearest neighbour and each processor contains a small element of local memory where immediate values are stored [68]. The control of data through a systolic array is marshalled by a synchronous clock [71], [72], [73]. Figure 3.17 illustrates a systolic array applied for matrix QR decomposition [30], [68].

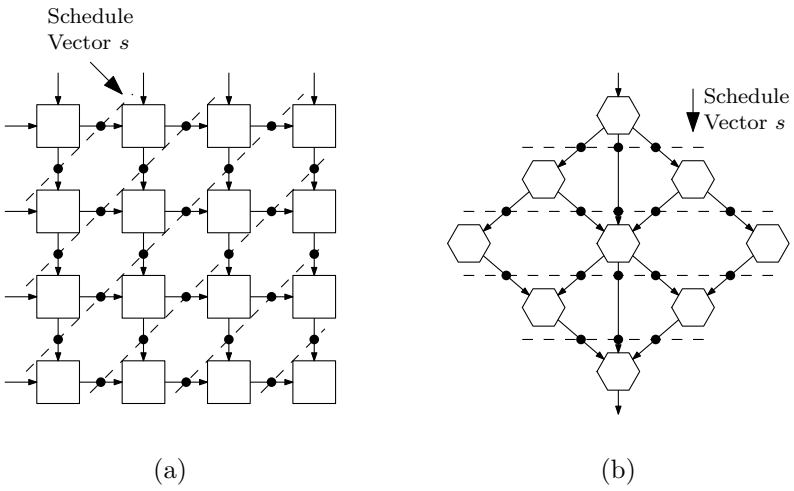



***Figure 3.16:*** *Linear systolic array architectures; (a) rectangular, and (b) hexagonal.*

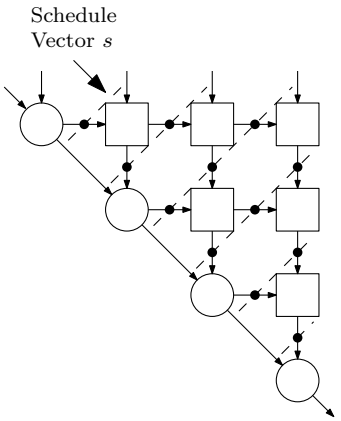



***Figure 3.17:*** *Triangular QR systolic array architecture.*

*Amdahl's Law* [74] provides a heuristic to calculate the theoretical maximum speed-up in latency a process can achieve by devoting parallel processing elements to its execution. A process's minimum theoretical latency is limited to the time required to execute the serial components of the process. Thus Amdahl's Law can be written as

$$S_{latency} = \frac{1}{(1-p) + \frac{p}{n}}, \tag{3.118}$$

where $S_{latency}$ is the theoretical speed-up in latency of the execution of the process, $p$ is the proportion of the process that can be made parallel, and $n$ is the number of parallel processing elements.

### 3.3.3 FPGA as a DSP Platform

Although signal processing is usually associated with digital signal processors, it is becoming increasingly evident that FPGAs are taking over as the platform of choice in the implementation of high-performance, high-precision signal processing [69]. As FPGA technology has evolved, the devices have increasingly come to be regarded as a complete system platform in themselves, including a DSP platform solution [68]. The emergence

of FPGA as a DSP platform was accelerated by the application of distributed arithmetic (DA) techniques [75], allowing implementations of typical DSP functionality realised from LUT-based / adder constructs considerable performance gains [68]. However the increase in architecture complexity due to technology evolution also heralds a growing gap in the scope offered by FPGA technology and the designer's ability to develop solutions efficiently using available tools. Some of the key issues that exist include; understanding how to map DSP functionality into FPGA devices, design languages, the development and the use of *Intellectual Property* (IP) cores, and design flow [68].

In the SoC / VLSI paradigm the engineer has the ability to create an architecture that can ultimately match the DSP system's performance requirements. However the practical limitations of developing an ultimate architecture inform the design approaches adopted, the reuse of existing architectural styles, and the utilisation of existing building DSP building blocks [68]. Thus the ultimate implementation is compromised for a limited range of functionality to ensure the creation of a system within a reasonable time frame. FPGA technologies are becoming more designer-and-implementation friendly, and vendors supplying more resources and offering new design-flows to aid in reducing system design and implementation time [69].

Specifically, utilising FPGA technology as a DSP platform allows the ability to scale adder word-length with application, where on traditional DSP platforms this would be fixed. Thus ripple-carry adder structures, composed of the concatenation of $N$ 1-bit adder structures, are offered as a dedicated resource on many FPGA architectures and compliments variable word length arithmetic [68] [69]. IP cores are also provided by vendors to assist with traditional functions such as FIR filtering and FFT computation [69].

Also, vendors are aware that one DSP solution does not fit all applications. Thus the range of FPGA technologies and DSP system development solutions are becoming granular across the spectrum of potential DSP applications. By offering variable-precision, and even both performance and high-precision capabilities like single or double-precision floating-point function blocks in the high-end devices, there are devices available that cater to almost any DSP application, within the vendors product catalogue [69]. Therefore FPGA technology has become more than just a viable DSP application platform worthy of consideration during system design and development.

## 3.4 Chaotic and Nonlinear Systems

Chaotic and nonlinear systems theory has become a vast and diverse field of interest since its accidental discovery by Lorenz. However, as with any vast and diverse field, so too is the jargon and fundamental concepts that define the field. It is difficult to describe one concept without presenting another, and is certainly the case in the reviewed literature. Thus to succinctly summarise and review the literature the author has presented these

concepts recursively, requiring the reader to adopt nonlinear and recursive reading patterns for the section. One can interpret this as a practical demonstration of some of the techniques and methods used within the field of chaotic and nonlinear systems theory.

### 3.4.1 Qualification and Quantification of Chaos

Chaos is the aperiodic, long-term behaviour of a bounded, deterministic system that exhibits *sensitive dependence on initial conditions* [76]. This sensitivity to initial conditions is quantified by calculating a system's spectrum of *Lyapunov exponents*, the commonly used measure of local system stability. The presence of a positive Lyapunov exponent indicates chaotic behaviour. The value of a positive Lyapunov exponent quantifies how chaotic a system's behaviour is.

The sum of the chaotic system's Lyapunov exponents will always be a negative number [76]. This indicates that even though the system is inherently unstable due to the presence of a positive Lyapunov exponent, $\lambda$, the stretching and folding of some trajectory within the system's state-space confines the trajectory's orbits to a finite and bounded phase-space. The Lyapunov exponents, or Lyapunov spectrum of a system, is a measure of the exponential separation, or stretching and folding, of neighbouring points in the evolution of a system's state-space portrait [77]. That is, two very similar initial values or neighbouring points on a trajectory in state-space will separate exponentially through the evolution of their respective orbits, as defined by the dynamics of the system.

### 3.4.2 Chaotic Oscillators

Figure 3.18 shows the state-space portrait of the Lorenz Attractor [18], defined by the following set of differential equations

$$\dot{x} = P(y - x), \tag{3.119}$$

$$\dot{y} = -xz + Rx - y, \tag{3.120}$$

$$\dot{z} = xy - Bz, \tag{3.121}$$

where $\dot{x}, \dot{y}$, and $\dot{z}$ are the derivatives of $x, y$, and $z$ with respect to time, $P, R$, and $B$ are adjustable system parameters, for some arbitrary initial conditions. Parameter values of $P = 10$, $R = 28$, and $B = 8/3$ are used to produce chaos and the state-space portrait shown in Fig. 3.18 [76].

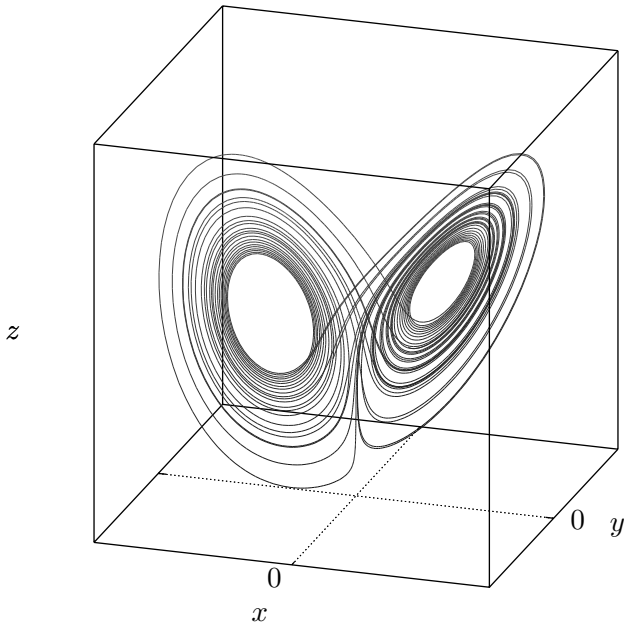



***Figure 3.18:*** *State-space portrait of the Lorenz attractor for $R = 28$, $P = 10$, $B = 8/3$, and some arbitrary initial conditions.*

Another well-understood chaotic system is defined by the *Mackey-Glass equation* [78],

$$\dot{x} = \frac{ax_\tau}{1 + x_\tau^c} - bx, \tag{3.122}$$

where $x_\tau = x(t - \tau)$ is the value of $x$ at time $t - \tau$. The system can be written in discrete form as

$$x[n+1] = x[n] + \frac{\tau}{N}\left(\frac{ax[n-N]}{1 + x^c[n-N]} - bx[n]\right). \tag{3.123}$$

The solution of Eq. 3.123 with parameters $a = 0.2$, $b = 0.1$, and $c = 10$ is chaotic for $\tau \gtrsim 16.8$; Fig. 3.19 shows the state-space portrait of the Mackey-Glass attractor for these parameters with $\tau = 23$ and $N = 1 \times 10^4$ [76].

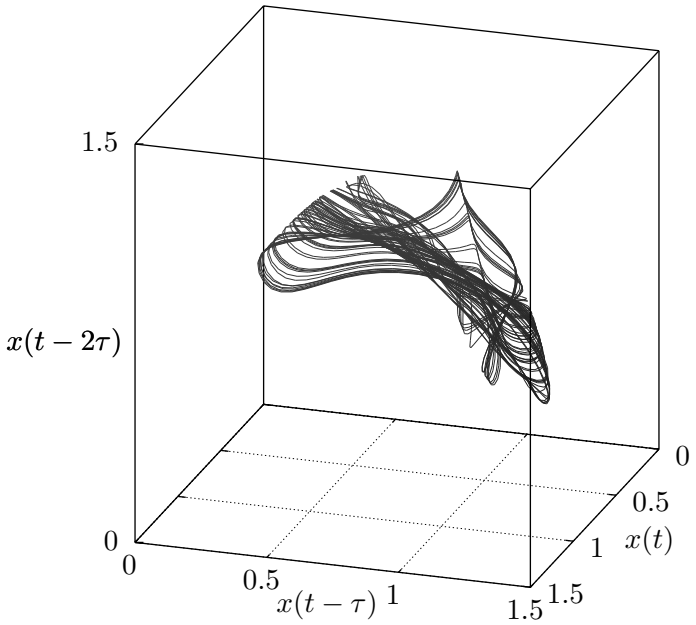



***Figure 3.19:*** *State-space portrait of the Mackey-Glass attractor for* $a = 0.2$*,* $b = 0.1$*,* $c = 10$*, and* $\tau = 23$*.*

Albers et-al [79] proposed a chaotic oscillator comprised of a *single-layer feed-forward* ANN given by

$$x[n+1] = \sum_{i-1}^{N} b_i \tanh\left(a_{i0} + \sum_{j=1}^{D} a_{ij} x[n-j]\right), \tag{3.124}$$

where $N$ is the number of neurons, $D$ is the number of time-lags, delay-line length, or input-vector dimension, coefficients $b_i$ are chosen from a random distribution uniform over $0 \leq b_i < 1$ and rescaled so the sum of their squares is $N$, and the connection-weight coefficients $a_{ij}$ are chosen from a random Gaussian distribution with zero mean and standard deviation $s$ [76].

By increasing the dimension of the input vector, and, by varying the standard deviation $s$ of the connection weights, the probability that the Albers et-al oscillator exhibits chaos can be increased from 0 to 1; a standard deviation of $s = 8$ ensures the earliest onset of chaos as $D$ is increased over the *Monte-Carlo method*-tested range of $1 \leq s \leq 128$. Figure 3.20 illustrates the architectural overview of the Albers et-al ANN chaotic oscillator.

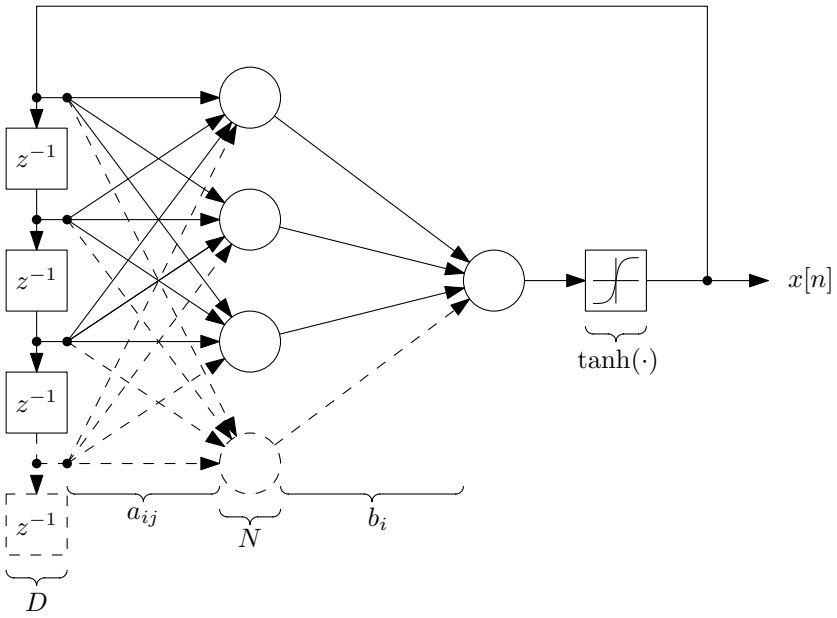



***Figure 3.20:*** *Albers et-al ANN chaotic oscillator architecture.*

### 3.4.3 State-space Embedding and State-space Reconstruction

Often the differential equations describing a system's dynamics are unknown; all that is known about a system is a set of periodically sampled measurements in the form of some single-dimensional time series.

Takens and Mañé's *embedding theorem* [80] enables the reconstruction of a system's state-space portrait from a measured time series. That is, from a set of scalar observations $s(t_0 + n\tau_s) = s[n]$, using $s[n]$ and its time delays $s[n + kT]$, vectors can be constructed in $d$-dimensional space

$$\bar{\boldsymbol{y}}[n] = \{s[n], s[n+T], s[n+2T], \cdots, s[n+T(d-1)]\} \tag{3.125}$$

that form a complete reconstruction of the system's state-space portrait. The theorem states that if a scalar quantity $h(\cdot)$ of some vector function of the dynamical variables $\bar{\boldsymbol{g}}(\bar{\boldsymbol{x}}[n])$ can be observed, then the geometric structure of the multivariate dynamics can be *unfolded* from the set of scalar measurements $h(\bar{\boldsymbol{g}}(\bar{\boldsymbol{x}}[n]))$ in a space spanned by new vectors with components consisting of $h(\cdot)$ applied to powers of $\bar{\boldsymbol{g}}(\bar{\boldsymbol{x}}[n])$. With appropriately smooth functions $h(\cdot)$ and $\bar{\boldsymbol{g}}(\bar{\boldsymbol{x}}[n])$, and if $d$ is large enough, the properties of the unknown multivariate dynamical variables $\bar{\boldsymbol{x}}[n]$ at the source of the chaos are reproduced without ambiguity in the new space of vector $\bar{\boldsymbol{y}}[n]$. By choosing $h(\cdot)$ and $\bar{\boldsymbol{g}}(\bar{\boldsymbol{x}}[n])$ carefully, state-space embedding can be carried out directly from the observed data. Letting the general scalar function equal the observed scalar variable

$$h(\bar{\boldsymbol{x}}[n]) = s[n], \tag{3.126}$$

and

$$\bar{\boldsymbol{g}}^{T_k}(\bar{\boldsymbol{x}}[n]) = \bar{\boldsymbol{x}}[n+T_k]), \tag{3.127}$$

then the components of $\bar{\boldsymbol{y}}[n]$ take the form

$$\bar{\boldsymbol{y}}[n] = \{s[n], s[n+T_1], s[n+T_2], \cdots, s[n+T_{d-1}]\}. \tag{3.128}$$

By setting $T_k = kT$, the time lags $T_k$ as integer multiples of a common lag $T$, the data vectors $\bar{\boldsymbol{y}}[n]$ become

$$\bar{\boldsymbol{y}}[n] = \{s[n], s[n+T], s[n+2T], \cdots, s[n+T(d-1)]\} \tag{3.129}$$

What remains is choosing a time lag $T$ and a dimension $d$ for the state-space embedding. Methods of choosing appropriate values for the time delay $T$ and dimension $d$ follow.

The choice if $T$ and $d$ enables the *unfolding* of the one-dimensional state-space portrait into a $d$-dimensional state-space portrait, thus revealing the system's dynamics. The observed system response $s[n]$ is by definition the nonlinear combination of initial conditions and dynamical variables, or, in a sense, a nonlinear projection of the system's state-space portrait onto one dimension. The goal is to choose $T$ and $d$ such that the projection down to one-dimensional space is adequately *undone*.

One proposed prescription for a suitable time delay $T$ is to find the first minimum of the autocorrelation $r_{ss}[m]$ of the time series $s[n]$. This is the optimum *linear* choice, from the point of view of predictability, in a least-squares sense, of $s[n+T]$ from knowledge of $s[n]$ [81]. However this is not such a good choice from a nonlinear perspective.

By regarding the chaotic process as a generator of information, one can apply Shannon's notion of mutual information [82]. Fraser [83], [84] proposed that $T$ be chosen as the first minimum of the average mutual information, of the time series $s[n]$. The average mutual information between two measurements, or the amount in bits learned by measurements of $s[n]$ through measurements of $s[n+T]$ is

$$I[T] = \sum_{s[n],s[n+T]} P(s[n], s[n+T]) \log_2 \left[ \frac{P(s[n], s[n+T])}{P(s[n])P(s[n+T])} \right] \tag{3.130}$$

where $P(s[n])$ and $P(s[n+T])$ are the individual probability densities $s[n]$ and its time lags, and $P(s[n], s[n+T])$ is the joint probability density for $s[n]$ and its time lag.

All that now remains is determining the integer global *embedding dimension* $d$, or more specifically $d_E$, where there are sufficient coordinates to unfold observed orbits from self overlaps arising from the projection of the attractor to a lower dimensional space. Note that $d_E$ is a *global* dimension and may well differ from the local dimension of the underlying dynamics [81].

The embedding theorem [80] also states that if the dimension of the attractor defined by the orbits is $d_A$, the *attractor dimension* of potentially some non-integer value, then an integer embedding dimension $d_E$ where $d_E > 2d_A$ will certainly unfold the attractor in $d_E$ dimensional space. This is not necessarily the minimum embedding dimension for unfolding, only an indication to stop adding dimension components to the time delay vector [81].

A test that determines and guarantees an appropriate unfolding of the attractor in $d$ dimensional space is the method of *False Nearest Neighbours* [85] [76]. Suppose a state-space reconstruction in dimension $d$ with data vectors as shown in Equation 3.129, constructed using the time delay suggested by average mutual information. Examine the nearest neighbour in state-space of the vector $\bar{\boldsymbol{y}}[n]$ with some time label $k$. This will be the vector

$$\bar{\boldsymbol{y}}^{NN}[k] = \{s^{NN}[k], s^{NN}[k+T], s^{NN}[k+2T], \cdots, s^{NN}[k+T(d-1)]\}. \tag{3.131}$$

If the vector $\bar{\boldsymbol{y}}^{NN}[k]$ is truly a neighbour of $\bar{\boldsymbol{y}}[n]$ then it came to the neighbourhood of $\bar{\boldsymbol{y}}[n]$ through dynamical origins. If the vector $\bar{\boldsymbol{y}}^{NN}[k]$ is a *false* neighbour of $\bar{\boldsymbol{y}}[n]$, having arrived in its neighbourhood by projection from a higher dimension because the present dimension $d$ does not unfold the attractor, then by going to the next dimension $d+1$ this false neighbour may be moved out of the neighbourhood of $\bar{\boldsymbol{y}}[n]$ [81]. By observing every data point $\bar{\boldsymbol{y}}[n]$ and testing for the dimension that removes all false neighbours, all intersections of orbits is sequentially removed, and an embedding dimension $d_E$ that unfolds the attractor is identified [85].

By comparing the distance between the vectors $\bar{\boldsymbol{y}}[n]$ and $\bar{\boldsymbol{y}}^{NN}[k]$ in dimension $d$ with the distance between the the same vectors in dimension $d+1$ it is trivial to establish which are true neighbours and which are false. The Euclidian distance between the nearest neighbour points as seen in dimension $d$ is given by

$$R_d = \sqrt{\sum_{m=1}^{d} \Big(s[n+T(m-1)] - s^{NN}[k+T(m-1)]\Big)^2}. \tag{3.132}$$

The criterion for falseness, using the distance between points when seen in dimension $d+1$ relative to the distance in dimension $d$, is given by

$$R_T < \frac{\sqrt{\Big(s[n+dT] - s^{NN}[k+dT]\Big)^2}}{R_d}, \tag{3.133}$$

where $R_T$ is some threshold value; Arbarbanel [81] recommends $R_T = 15$, however the value is not critical [76].

# Chapter 4

# SVM System Architectures and Scientific Method

This chapter provides all SVM system design, development, and implementation details, and, an overview of the experimental procedures designed and conducted as part of this research work. This chapter also includes system architecture design and implementations.

Three novel SVM training strategies where developed as part of this body of research; these training strategies are presented in Section 4.1. The design and implementation of the SVM test-rig system architecture architecture is presented in Section 4.2. The design and implementation of the SVM training and function-evaluation DSP pipelines is presented in Section 4.3. Finally, as informed by the reviewed literature of Chapter 3 and the design, implementation, and functional testing of the SVM systems, the scientific methodologies applied as part of this body of work is presented in Section 4.4.

## 4.1 SVM Training Strategies

Three SVM training architectures have been designed for both SVM classification and regression problems. The first is a brute-force non-iterative fixed-period SVM training method that has undergone successful peer-review and was subsequently published in the proceedings of the IEEE Industrial Electronics Society Conference 2013 (IECON13) [86]. The second and third architectures are designed based on the mapped Neural Network optimisation techniques, the Combined Exterior Penalty and Interior Penalty / Barrier Function method and the Augmented Lagrange Multiplier method respectively, presented by Cichocki and Unbehauen [37]. The continuous-time ANN-mapped optimisation techniques presented in Section 3.1.1.4 have been developed here in continuous time and then redefined as discrete time equivalent systems. All three training strategies have been implemented as software prototypes to study their feasibility as SVM training solutions.

### 4.1.1 Brute-force SVM Training

The brute-force non-iterative fixed-period SVM training method makes unreserved use of the potential parallelism viable by means of an FPGA-implementation. The method also exploits the fundamental mechanism of operation of the SVM as allured to in Eq. 3.41, and utilises knowledge gained by observing the optimisation constraints shown in Eq. 3.39 and Eq. 3.40 and using this knowledge to guide support vector and Lagrangian coefficient pair identification.

By performing the costly constrained optimisation operation, one learns all the vector and Lagrangian coefficient pairs $\{\bar{\boldsymbol{x}}_i, \alpha_i\}$ that maximise the objective function shown in Eq. 3.38 subject to the constraints shown in Eq. 3.39 and Eq. 3.40. However only support vectors are required to define the pair of supporting hyperplanes, and thus, accord the computation of the classification model $\hat{f}(\bar{\boldsymbol{x}})$; the pragmatic reality of the situation only calls for $n$ support vector and Lagrangian coefficient pairs $\{\bar{\boldsymbol{x}}_{sv^+}, \alpha_{sv^+}\}$ and $\{\bar{\boldsymbol{x}}_{sv^-}, \alpha_{sv^-}\}$ from each class for SVM model evaluation.

Observing the constraints shown in Eq. 3.39 and Eq. 3.40 one is compelled to conclude that all Lagrangian coefficients must be positive, and therefore $\alpha_i y_i$ must exist in equal and opposite pairs - equally valued Lagrangian coefficients pairs, one for each $+1$ and $-1$ class. This knowledge thus forms the fundamental basis of the training algorithm.

The training algorithm is subdivided into a prologue stage and four separate and distinct processing stages, as outlined in Fig. 4.1; the following provides an exposition of each of the algorithm's stages. A supplemental Optimal SVM Function-Model Evaluation stage, as shown in Section 4.3, is included here to provide training-process clarity and context.

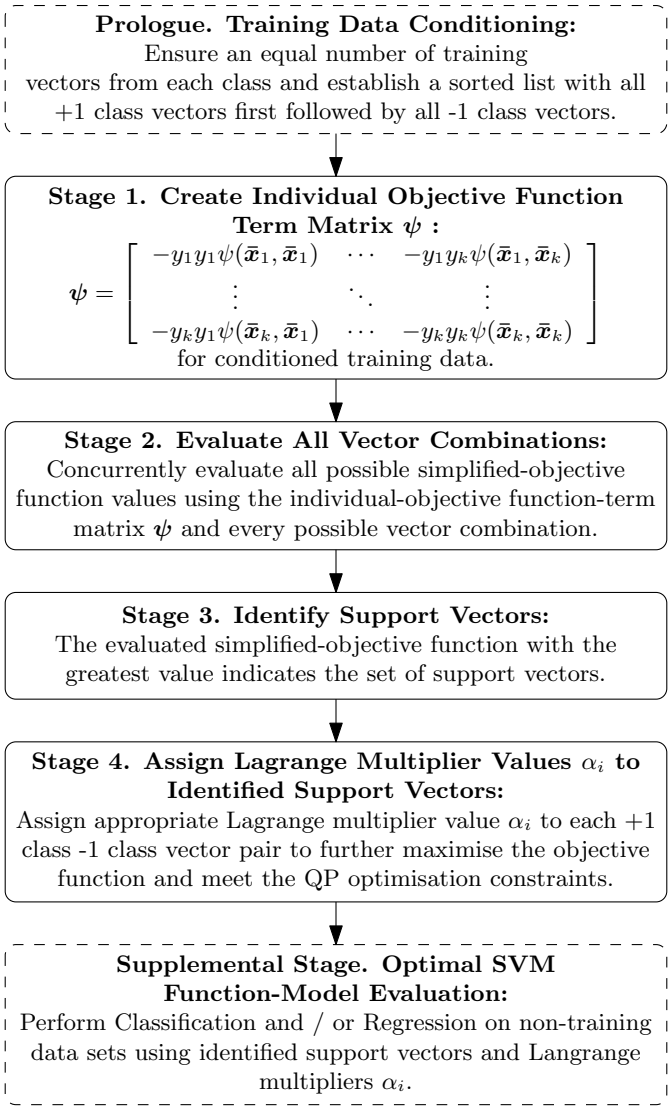



***Figure 4.1:*** *The non-iterative fixed-period SVM training algorithm including supplementary SVM optimal-function model evaluation stage for classification and / or regression.*

**Prologue. Training Data Conditioning.** For the algorithm to function as desired training data must be presented in a compatible form. For each training vector in +1 class, there must also exist a -1 class vector; there must be an equal number of +1 class and -1 class training vectors. Also, the vectors must be presented in a list ordered by class, with +1 class vectors first followed by -1 class vectors. The prologue can serve to either verify this training data format, ensure by deterministic algorithm this training data format requirements, or be excluded entirely conditional to some upstream external data-acquisition and conditioning system.

**Stage 1. Create Individual Objective Function Term Matrix $\boldsymbol{\psi}$.** Every potential simplified objective function term is calculated. The simplified objective function term is derived from the SVM classification objective function shown in Eq. 3.38 as follows. One

already knows from the optimisation constraints the term

$$\sum_{i=1}^{k} \alpha_i \tag{4.1}$$

will be positive, thus it is ignored. This leaves only the term

$$-\frac{1}{2} \sum_{i=1}^{k} \sum_{j=1}^{k} \alpha_i \alpha_j y_i y_j \psi(\bar{\boldsymbol{x}}_i, \bar{\boldsymbol{x}}_j). \tag{4.2}$$

The $\frac{1}{2}$ serves only to scale the final value of the term, thus in the name of efficiency it too is ignored. The Lagrange multiplier pair $\alpha_i \alpha_j$ are set to 1 to further simplify the expression. Thus the simplified objective function term takes the form

$$-y_i y_j \psi(\bar{\boldsymbol{x}}_i, \bar{\boldsymbol{x}}_j), \tag{4.3}$$

and the collection of all indexed training vector objective function term combinations form the individual objective function term matrix $\boldsymbol{\psi}$

$$\boldsymbol{\psi} = \begin{bmatrix} -y_1 y_1 \psi(\bar{\boldsymbol{x}}_1, \bar{\boldsymbol{x}}_1) & \cdots & -y_1 y_k \psi(\bar{\boldsymbol{x}}_1, \bar{\boldsymbol{x}}_k) \\ \vdots & \ddots & \vdots \\ -y_k y_1 \psi(\bar{\boldsymbol{x}}_k, \bar{\boldsymbol{x}}_1) & \cdots & -y_k y_k \psi(\bar{\boldsymbol{x}}_k, \bar{\boldsymbol{x}}_k) \end{bmatrix}. \tag{4.4}$$

Figure 4.2 illustrates matrix $\boldsymbol{\psi}$ in terms of the training vector classes. Subject to the kernel function $\psi(\bar{\boldsymbol{x}}_k, \bar{\boldsymbol{x}}_k)$ employed in the SVM system, terms on the upper-right diagonal of matrix $\boldsymbol{\psi}$ are repeated in the lower-left and thus need not be recalculated; this fact significantly impacts on processing and memory requirements of this stage.

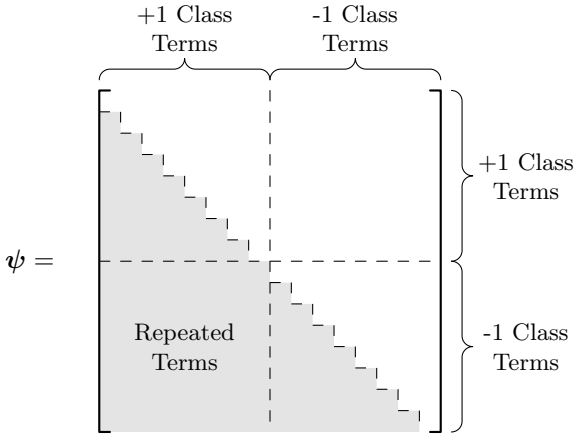



***Figure 4.2:*** *Individual objective function term matrix $\boldsymbol{\psi}$ illustrating the four class-combination quadrants, and when utilising an appropriate kernel, repeated terms that needn't be calculated.*

**Stage 2. Evaluate All Vector Combinations.** The sum of every vector combination possible is found by brute-force parallel computation of terms on the corresponding rows and columns the matrix $\boldsymbol{\psi}$. Figure 4.3 illustrates every possible vector combination of a

training data set of eight 2-dimensional vectors, thus there exists 2 support vectors for each class, and $\boldsymbol{\psi}$ is an $8\times8$ square matrix; the dark-grey boxes where each corresponding vector intersections illustrates the terms to be summed together to form one of the potential maximisations of the objective function shown in Eq. 3.41.

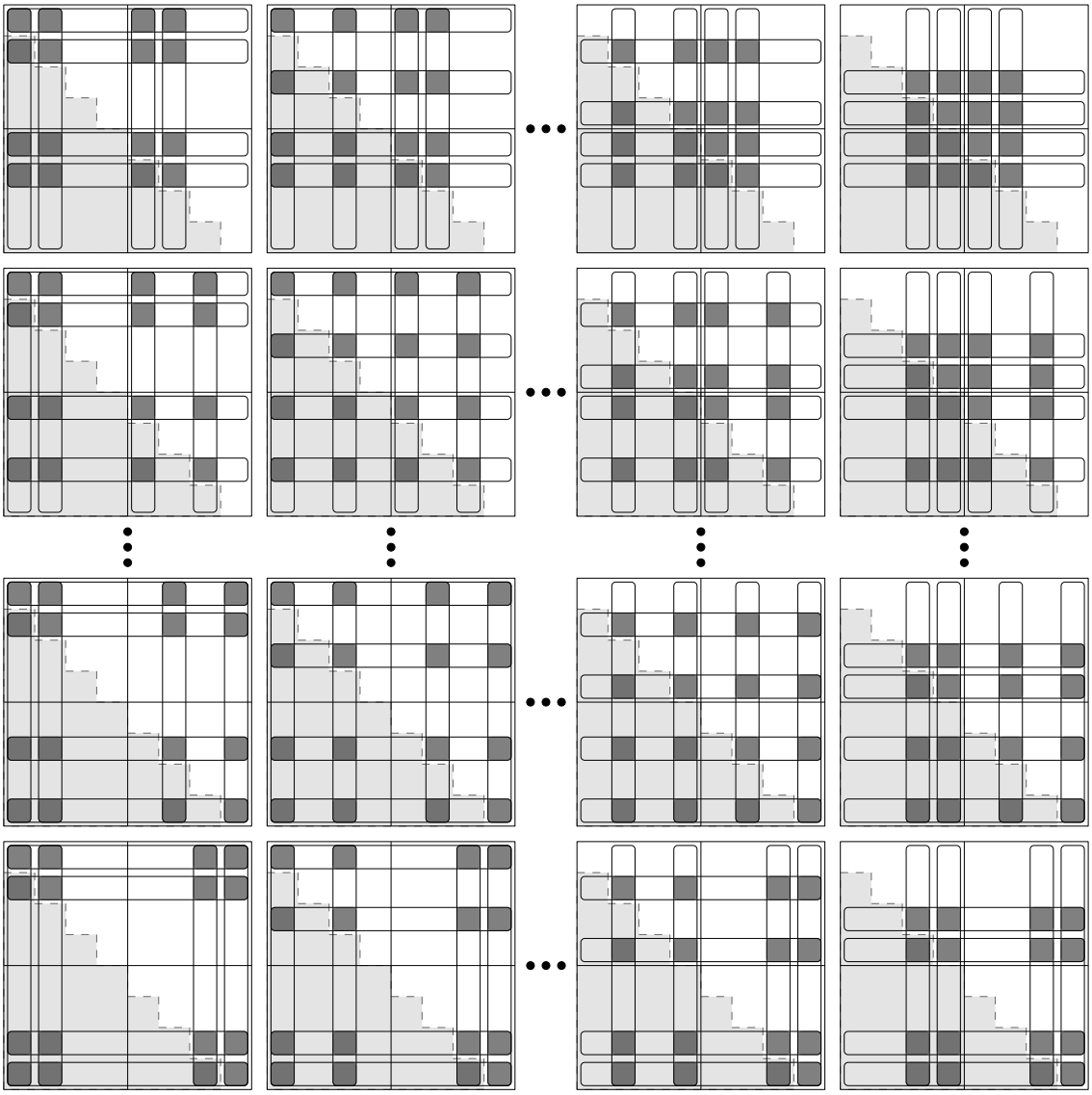


***Figure 4.3:*** *All vector combination patterns for an eight 2-dimensional vector training set; the dark-grey boxes where each corresponding vector intersections illustrates the terms to be summed to form one of the potential maximisations of the objective function.*

**Stage 3. Identify Support Vectors.** The maximum, or set of maximum values in the case of a maxima stalemate, are then deemed to indicate the set of possible support vectors. The dark-grey boxes shown in Fig. 4.3 corresponding to vector intersections are again summed together for each of the maximum cases, however this time the terms are modified according to a finite set of Lagrange multiplier coefficient combinations $\alpha_i\alpha_j$, where $\alpha_1 = 1, \alpha_2 = 2, \cdots, \alpha_n = n$ and $n$ is the number of support vectors. This set of calculations aims to resolve any maxima impasse and provide more insight into Lagrange coefficient allocation in Stage 4. If a maxima impasse is not resolved it is deemed there

exits more than one set of unique support vectors and therefore any arbitrary choice of the between the stalemate-factions will sufficiently serve.

**Stage 4. Assign Lagrange Multiplier Values $\alpha_i$ to Identified Support Vectors.** Observing the results of Stage 3. one can deduce which vectors of the training data receive non-zero Lagrangian coefficients $\alpha$ and the fraction of the cost-constant $C$ these values will take. Care must be taken to ensure that Lagrange multipliers values are allocated in pairs, where $\alpha^i_{sv^+} = \alpha^i_{sv^-}$, and the constraint shown in Eq. 3.39 holds true.

**Supplemental Stage. Optimal SVM Function-Model Evaluation.** Using the recently acquired Lagrange multiplier and support vector pairs $\{\bar{\boldsymbol{x}}^i_{sv^+}, \alpha^i_{sv^+}\}$ and $\{\bar{\boldsymbol{x}}^i_{sv^-}, \alpha^i_{sv^-}\}$, one can now evaluate the optimum SVM function-model shown in Fig. 3.41 on new non-training data; a general architectural structure for this stage is presented in Section 4.3.

Where appropriate signed unitary multiplication operations are replaced with appropriate *sign-bit modifying* digital-logic circuitry for both computational-performance enhancement and resource conservation.

Each parallel calculation of Stage 1. is carried out in two steps. The sign of each $y_i y_j$ class-coefficient pair and the kernel function $\psi(\bar{\boldsymbol{x}}_i, \bar{\boldsymbol{x}}_j)$ are evaluated for $i, j = 1, 2, \cdots, k$, where $k$ denotes the last training vector. The class-coefficients $y_i y_j$ are assigned a binary value, where

$$a_i = \begin{cases} 1 & \text{for} \quad -\text{ve } y_i \\ 0 & \text{for} \quad +\text{ve } y_i \end{cases}, \tag{4.5}$$

and the sign-bit $Y_{ij}$ of the pair is found, where

$$Y_{ij} = \overline{a_i \oplus a_j}. \tag{4.6}$$

Then the sign-bit of each kernel function evaluation is XORed with each corresponding $Y_{ij}$, thus effectively evaluating $-y_i y_j \psi(\bar{\boldsymbol{x}}_i, \bar{\boldsymbol{x}}_j)$ with minimal costly multiplication operations. Figure 4.4 provides an illustration of the Stage 1. architecture.

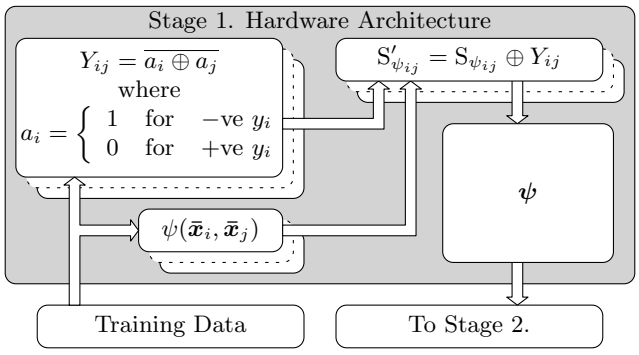



***Figure 4.4:*** *Stage 1. hardware architecture overview.*

Stages 2, 3, and 4 are are shown as architectural overviews in Fig. 4.5, Fig. 4.6, and

Fig. 4.7 respectively. Currently these stages use simple straight-forward logic and sorting techniques, thus no further exposition into the underlying architecture is provided.

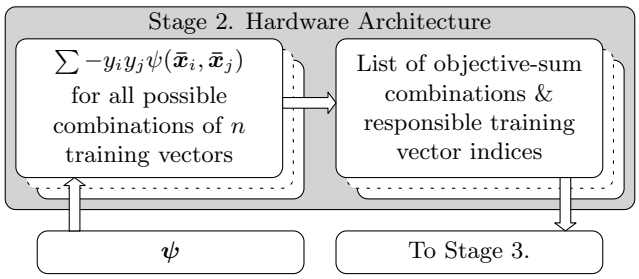



***Figure 4.5:*** *Stage 2. hardware architecture overview.*

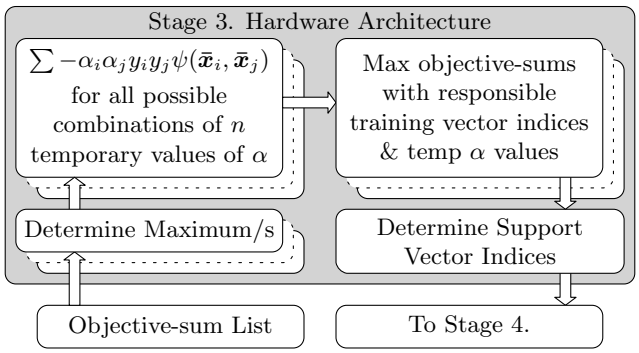



***Figure 4.6:*** *Stage 3. hardware architecture overview.*

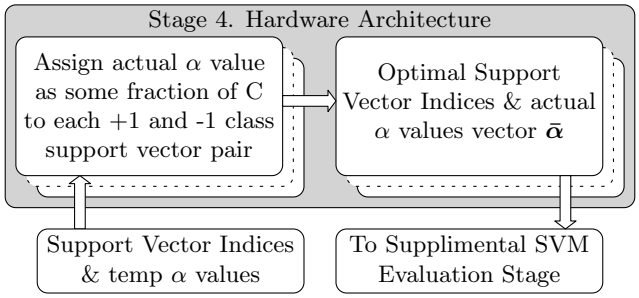



***Figure 4.7:*** *Stage 4. hardware architecture overview.*

Where appropriate, the DSP pipeline architectures presented in Fig. 4.29, Fig. 4.30, Fig. A.1, Fig. A.2, Fig. A.3, Fig. 4.31, and Fig. 4.32 can be used to construct the brute-force non-iterative fixed-period SVM classification training subsystem architecture.

As a proof-of-concept and verification of successful SVM training the architecture was modelled using MATLAB. By applying the trained SVM to test data-sets with the goal of correctly classifying the data, the SVM training scheme was thus validated as a feasible mechanism for SVM training. All parallel processes inherent in the design were modelled in a sequential manner.

The design was tested using a simple 2-dimensional linearly-separable problem train-

ing data-set utilising the dot-product kernel, the *no-kernel* variation of the SVM. The number of computations for Stage 1 to Stage 4 were estimated to be 765, 12960, 12960, and 4 operations respectively. The derived Lagrangian coefficients and support vectors were then applied to SVM function-model evaluation with a different linearly-separable testing data-set. Both training and testing linearly-separable data-sets are shown in Fig. 4.8. The linearly-separable testing data-set was classified with 100% accuracy.

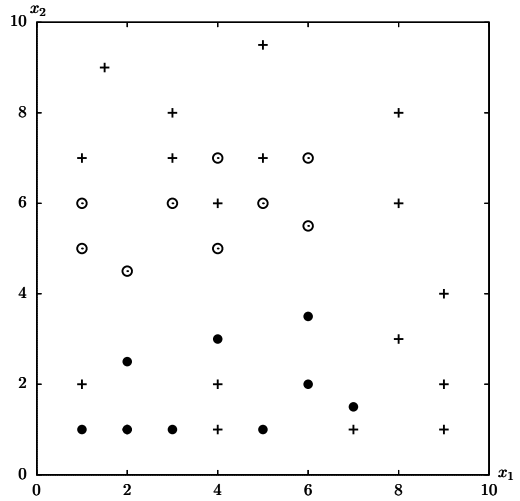



***Figure 4.8:*** *Simple linearly-separable problem datasets; +1 class and -1 class training data are shown as circles and dots respectively, testing data is shown as crosses.*

The design was also tested using a standard 2-dimensional XOR problem training data-set utilising a polynomial kernel. The number of computations for Stage 1 to Stage 4 were estimated to be 1224, 12960, 12960, and 4 operations respectively. The derived Lagrangian coefficients and support vectors were then applied to SVM function-model evaluation with a different XOR testing data-set. Both training and testing XOR data-sets are shown in Fig. 4.9. The XOR testing dataset was classified with 100% accuracy.

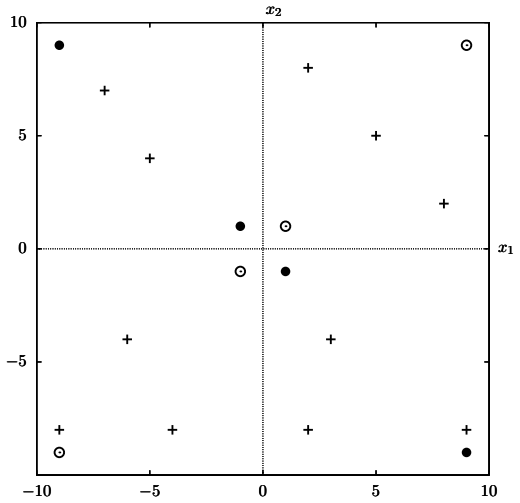



***Figure 4.9:*** *XOR problem datasets; +1 class and -1 class training data are shown as circles and dots respectively, testing data is shown as crosses.*

### 4.1.2 Combined Exterior Penalty and Barrier Function Optimisation

The combined exterior penalty function and interior penalty / barrier function optimisation technique, shown in Eq. 3.62, was developed for the SVM classification problem as follows. The soft-margin support vector model is trained by optimising the Lagrangian

$$\arg\max_{\bar{\boldsymbol{\alpha}}} \phi'(\bar{\boldsymbol{\alpha}}) = \arg\max_{\bar{\boldsymbol{\alpha}}} \left( \sum_{i=1}^{k} \alpha_i - \frac{1}{2} \sum_{i=1}^{k} \sum_{j=1}^{k} \alpha_i \alpha_j y_i y_j \psi(\bar{\boldsymbol{x}}_i, \bar{\boldsymbol{x}}_j) \right), \tag{4.7}$$

subject to the constraints

$$\sum_{i=1}^{k} \alpha_i y_i = 0, \tag{4.8}$$

$$\alpha_i \geq 0, \text{ and} \tag{4.9}$$

$$C - \alpha_i \geq 0, \tag{4.10}$$

where $i = 1, \cdots, k$ and $C$ is the cost constant.

Applying the extended interior approach the constrained optimisation problem shown in Eq. 4.7 , Eq. 4.8, Eq. 4.9 and Eq. 4.10 is converted into an unconstrained minimisation problem by constructing an energy function of the form

$$E(\bar{\boldsymbol{\alpha}}, \bar{\boldsymbol{\kappa}}) = \phi(\bar{\boldsymbol{\alpha}}) + \kappa_0 h^2(\bar{\boldsymbol{\alpha}}) + \frac{1}{\kappa_1} B_1(\bar{\boldsymbol{\alpha}}) + \frac{1}{\kappa_2} B_2(\bar{\boldsymbol{\alpha}}), \tag{4.11}$$

where

$$\phi(\bar{\boldsymbol{\alpha}}) = -\sum_{i=1}^{k} \alpha_i + \frac{1}{2} \sum_{i=1}^{k} \sum_{j=1}^{k} \alpha_i \alpha_j y_i y_j \psi(\bar{\boldsymbol{x}}_i, \bar{\boldsymbol{x}}_j), \tag{4.12}$$

$$h(\bar{\boldsymbol{\alpha}}) = \sum_{i=1}^{k} \alpha_i y_i, \tag{4.13}$$

where $\kappa_0 > 0, \kappa_1 > 0$, and $\kappa_2 > 0$, and where $B_1(\bar{\boldsymbol{\alpha}})$ and $B_2(\bar{\boldsymbol{\alpha}})$ are the extended barrier functions

$$B_1(\alpha_j) = \begin{cases} \dfrac{1}{\alpha_j}, & \text{if} \quad \alpha_j \geq \epsilon, \\ \dfrac{2\epsilon - \alpha_j}{\epsilon^2}, & \text{if} \quad \alpha_j < \epsilon, \quad \text{and} \end{cases} \tag{4.14}$$

$$B_2(\alpha_j) = \begin{cases} \dfrac{1}{C - \alpha_j}, & \text{if} \quad C - \alpha_j \geq \epsilon, \\ \dfrac{2\epsilon - C + \alpha_j}{\epsilon^2}, & \text{if} \quad C - \alpha_j < \epsilon, \end{cases} \tag{4.15}$$

where $\epsilon$ is a small positive number which determines the transition from the exterior extended penalty to the interior penalty functions.

Applying standard gradient descent technique a system of differential equations is con-

structed and the local minimum of the energy function shown in Eq. 4.11 is found:

$$\frac{\mathrm{d}\bar{\boldsymbol{\alpha}}}{\mathrm{d}t} = -\boldsymbol{\mu}\nabla_{\bar{\boldsymbol{\alpha}}} E(\bar{\boldsymbol{\alpha}}, \bar{\boldsymbol{\kappa}}) + \bar{\boldsymbol{\nu}}(t), \tag{4.16}$$

for the initial conditions $\bar{\boldsymbol{\alpha}}(0) = \bar{\boldsymbol{\alpha}}^{(0)}$, and where $\boldsymbol{\mu} = \mathrm{diag}(\mu_1, \cdots, \mu_n)$, $\mu_j > 0$ for $j = 1, \cdots, n$, where typically $\mu_j = \mu = 1/\tau$ for all $j$ where $\tau$ is the integration time constant, and $\bar{\boldsymbol{\nu}}(t)$ is an uncorrelated white noise source with zero mean and variance decreasing in time, or, an array of one-dimensional chaotic oscillation sources. Thus

$$\frac{\mathrm{d}\alpha_j}{\mathrm{d}t} = -\mu_j \left( \frac{\partial \phi(\bar{\boldsymbol{\alpha}})}{\partial \alpha_j} + \kappa_0 h(\bar{\boldsymbol{\alpha}}) \frac{\partial h(\bar{\boldsymbol{\alpha}})}{\partial \alpha_j} + \frac{1}{\kappa_1} \frac{\partial B_1(\bar{\boldsymbol{\alpha}})}{\partial \alpha_j} + \frac{1}{\kappa_2} \frac{\partial B_2(\bar{\boldsymbol{\alpha}})}{\partial \alpha_j} \right) + \nu_j(t), \tag{4.17}$$

for $\alpha_j(0) = \alpha_j^{(0)}$.

Equation 4.17 is finally rewritten as discrete-time equivalent

$$\alpha_j[n+1] = \alpha_j[n] - \mu_j \Bigg[ \phi_{\alpha_j}(\bar{\boldsymbol{\alpha}}[n]) + \kappa_0 y_j h(\bar{\boldsymbol{\alpha}}[n]) + \frac{1}{\kappa_1} B_{1\,\alpha_j}(\alpha_j[n]) + \frac{1}{\kappa_2} B_{2\,\alpha_j}(\alpha_j[n]) \Bigg] + \nu_j[n], \tag{4.18}$$

for $\alpha_j[0] = \alpha_j^{[0]}$, where

$$\phi_{\alpha_j}(\bar{\boldsymbol{\alpha}}) = \frac{1}{2} \sum_{i=1}^{k} \alpha_i[n] y_i y_j \psi(\bar{\boldsymbol{x}}_i, \bar{\boldsymbol{x}}_j) - 1, \tag{4.19}$$

$$B_{1\,\alpha_j}(\alpha_j) = \begin{cases} -\dfrac{1}{\alpha_j^2}\,, & \text{if} \quad \alpha_j \geq \epsilon, \\ -\dfrac{1}{\epsilon^2}\,, & \text{if} \quad \alpha_j < \epsilon, \quad \text{and} \end{cases} \tag{4.20}$$

$$B_{2\,\alpha_j}(\alpha_j) = \begin{cases} \dfrac{1}{(C-\alpha_j)^2}\,, & \text{if} \quad C - \alpha_j \geq \epsilon, \\ \dfrac{1}{\epsilon^2}\,, & \text{if} \quad C - \alpha_j < \epsilon. \end{cases} \tag{4.21}$$

Figure 4.10 illustrates an implementable functional block diagram of the ANN shown in Eq. 4.18 for one optimisation variable $\alpha_j$; for diagram clarity only optimisation variable inputs are shown.

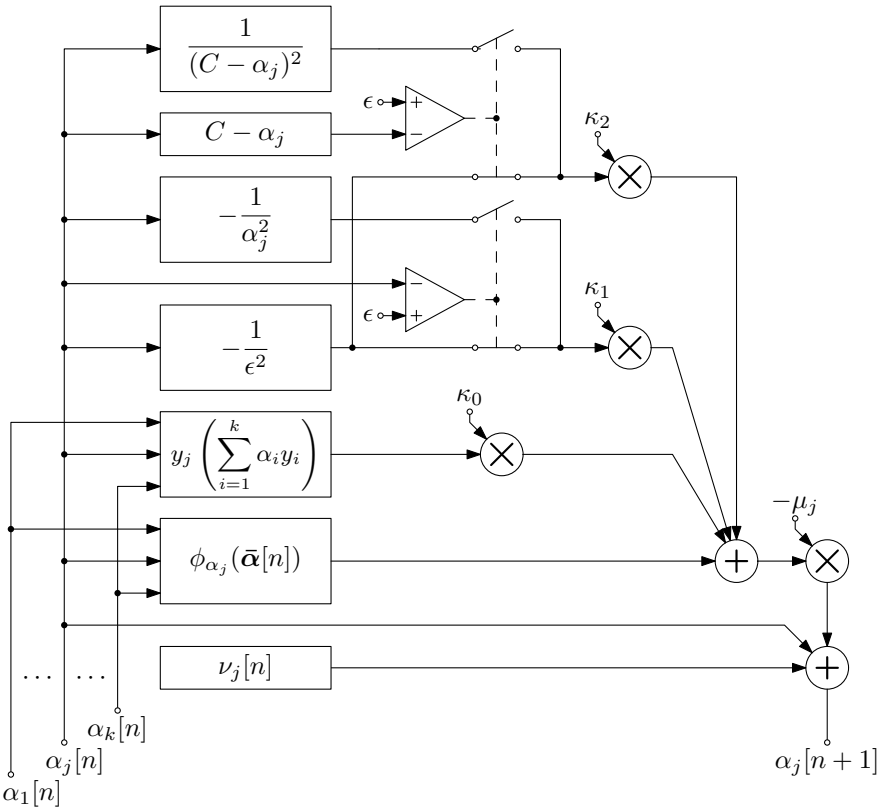



***Figure 4.10:*** *Functional block-diagram of the* ANN*-mapped combined exterior penalty function and interior penalty / barrier function optimisation technique for* SVM *classification as defined in Eq. 4.18.*

The combined exterior penalty function and interior penalty / barrier function optimisation technique, shown in Eq. 3.62, was developed for the SVM regression problem as follows. The soft-margin support vector model for regression is trained by optimising the Lagrangian

$$\underset{\bar{\boldsymbol{\alpha}}^{\bullet},\bar{\boldsymbol{\alpha}}^{\circ}}{\arg\max}\,\phi'(\bar{\boldsymbol{\alpha}}^{\bullet},\bar{\boldsymbol{\alpha}}^{\circ}) = \underset{\bar{\boldsymbol{\alpha}}^{\bullet},\bar{\boldsymbol{\alpha}}^{\circ}}{\arg\max}\left(-\frac{1}{2}\sum_{i=1}^{k}\sum_{j=1}^{k}(\alpha_i^{\bullet}-\alpha_i^{\circ})(\alpha_j^{\bullet}-\alpha_j^{\circ})\psi(\bar{\boldsymbol{x}}_i,\bar{\boldsymbol{x}}_j) + \sum_{i=1}^{k} y_i(\alpha_i^{\bullet}-\alpha_i^{\circ}) - \eta\sum_{i=1}^{k}(\alpha_i^{\bullet}+\alpha_i^{\circ})\right), \quad (4.22)$$

subject to the constraints

$$\sum_{i=1}^{k}(\alpha_i^{\bullet}-\alpha_i^{\circ}) = 0, \quad (4.23)$$

$$\alpha_i^{\circ} \geq 0, \text{ and} \quad (4.24)$$

$$C-\alpha_i^{\bullet} \geq 0, \quad (4.25)$$

where $i = 1, \cdots, k$ and $C$ is the cost constant.

Applying the extended interior approach the constrained optimisation problem shown

in Eq. 4.22 , Eq. 4.23, Eq. 4.24 and Eq. 4.25 is converted into an unconstrained minimisation problem by constructing an energy function of the form

$$E(\bar{\boldsymbol{\alpha}}^{\bullet}, \bar{\boldsymbol{\alpha}}^{\circ}, \bar{\boldsymbol{\kappa}}) = \phi(\bar{\boldsymbol{\alpha}}^{\bullet}, \bar{\boldsymbol{\alpha}}^{\circ}) + \kappa_0 h^2(\bar{\boldsymbol{\alpha}}^{\bullet}, \bar{\boldsymbol{\alpha}}^{\circ}) + \frac{1}{\kappa_1} B_1(\bar{\boldsymbol{\alpha}}^{\bullet}, \bar{\boldsymbol{\alpha}}^{\circ}) + \frac{1}{\kappa_2} B_2(\bar{\boldsymbol{\alpha}}^{\bullet}, \bar{\boldsymbol{\alpha}}^{\circ}), \tag{4.26}$$

where

$$\phi(\bar{\boldsymbol{\alpha}}^{\bullet}, \bar{\boldsymbol{\alpha}}^{\circ}) = \frac{1}{2} \sum_{i=1}^{k} \sum_{j=1}^{k} (\alpha_i^{\bullet} - \alpha_i^{\circ})(\alpha_j^{\bullet} - \alpha_j^{\circ}) \psi(\bar{\boldsymbol{x}}_i, \bar{\boldsymbol{x}}_j) - \sum_{i=1}^{k} y_i (\alpha_i^{\bullet} - \alpha_i^{\circ}) + \eta \sum_{i=1}^{k} (\alpha_i^{\bullet} + \alpha_i^{\circ}), \tag{4.27}$$

$$h(\bar{\boldsymbol{\alpha}}^{\bullet}, \bar{\boldsymbol{\alpha}}^{\circ}) = \sum_{i=1}^{k} (\alpha_i^{\bullet} - \alpha_i^{\circ}), \tag{4.28}$$

where $\kappa_0 > 0, \kappa_1 > 0$, and $\kappa_2 > 0$, and where $B_1(\bar{\boldsymbol{\alpha}}^{\bullet}, \bar{\boldsymbol{\alpha}}^{\circ})$ and $B_2(\bar{\boldsymbol{\alpha}}^{\bullet}, \bar{\boldsymbol{\alpha}}^{\circ})$ are the extended barrier functions

$$B_1(\alpha_j^{\bullet}, \alpha_j^{\circ}) = \begin{cases} \dfrac{1}{\alpha_j^{\circ}}, & \text{if} \quad \alpha_j^{\circ} \geq \epsilon, \\ \dfrac{2\epsilon - \alpha_j^{\circ}}{\epsilon^2}, & \text{if} \quad \alpha_j^{\circ} < \epsilon, \quad \text{and} \end{cases} \tag{4.29}$$

$$B_2(\alpha_j^{\bullet}, \alpha_j^{\circ}) = \begin{cases} \dfrac{1}{C - \alpha_j^{\bullet}}, & \text{if} \quad C - \alpha_j^{\bullet} \geq \epsilon, \\ \dfrac{2\epsilon - C + \alpha_j^{\bullet}}{\epsilon^2}, & \text{if} \quad C - \alpha_j^{\bullet} < \epsilon, \end{cases} \tag{4.30}$$

where $\epsilon$ is a small positive number which determines the transition from the exterior extended penalty to the interior penalty functions.

Applying standard gradient descent technique a system of differential equations is constructed and the local minimum of the energy function shown in Eq. 4.11 is found:

$$\frac{\mathrm{d}\bar{\boldsymbol{\alpha}}^{\bullet}}{\mathrm{d}t} = -\boldsymbol{\mu} \nabla_{\bar{\boldsymbol{\alpha}}^{\bullet}} E(\bar{\boldsymbol{\alpha}}^{\bullet}, \bar{\boldsymbol{\alpha}}^{\circ}, \bar{\boldsymbol{\kappa}}) + \bar{\boldsymbol{\nu}}_{\alpha^{\bullet}}(t), \tag{4.31}$$

$$\frac{\mathrm{d}\bar{\boldsymbol{\alpha}}^{\circ}}{\mathrm{d}t} = -\boldsymbol{\rho} \nabla_{\bar{\boldsymbol{\alpha}}^{\circ}} E(\bar{\boldsymbol{\alpha}}^{\bullet}, \bar{\boldsymbol{\alpha}}^{\circ}, \bar{\boldsymbol{\kappa}}) + \bar{\boldsymbol{\nu}}_{\alpha^{\circ}}(t), \tag{4.32}$$

for the initial conditions $\bar{\boldsymbol{\alpha}}(0) = \bar{\boldsymbol{\alpha}}^{(0)}$ and where $\boldsymbol{\mu} = \mathrm{diag}(\mu_1, \cdots, \mu_n)$ and $\boldsymbol{\rho} = \mathrm{diag}(\rho_1, \cdots, \rho_n)$, $\mu_j > 0$ and $\rho_j > 0$ for $j = 1, \cdots, n$, where typically $\mu_j = \mu = \rho_j = \rho = 1/\tau$ for all $j$ where $\tau$ is the integration time constant, and $\bar{\boldsymbol{\nu}}_{\alpha^{\bullet}}(t)$ and $\bar{\boldsymbol{\nu}}_{\alpha^{\circ}}(t)$ are uncorrelated white noise sources with zero mean and variance decreasing in time, or, are arrays of one-dimensional chaotic oscillation sources. Thus

$$\frac{\mathrm{d}\alpha_j^{\bullet}}{\mathrm{d}t} = -\mu_j \left( \frac{\partial \phi(\bar{\boldsymbol{\alpha}}^{\bullet}, \bar{\boldsymbol{\alpha}}^{\circ})}{\partial \alpha_j^{\bullet}} + \kappa_0 h(\bar{\boldsymbol{\alpha}}^{\bullet}, \bar{\boldsymbol{\alpha}}^{\circ}) \frac{\partial h(\bar{\boldsymbol{\alpha}}^{\bullet}, \bar{\boldsymbol{\alpha}}^{\circ})}{\partial \alpha_j^{\bullet}} + \frac{1}{\kappa_1} \frac{\partial B_1(\bar{\boldsymbol{\alpha}}^{\bullet}, \bar{\boldsymbol{\alpha}}^{\circ})}{\partial \alpha_j^{\bullet}} + \frac{1}{\kappa_2} \frac{\partial B_2(\bar{\boldsymbol{\alpha}}^{\bullet}, \bar{\boldsymbol{\alpha}}^{\circ})}{\partial \alpha_j^{\bullet}} \right) + \nu_{\alpha_j^{\bullet}}(t), \tag{4.33}$$

and

$$\frac{\mathrm{d}\alpha_j^{\circ}}{\mathrm{d}t} = -\rho_j \left( \frac{\partial \phi(\bar{\boldsymbol{\alpha}}^{\bullet}, \bar{\boldsymbol{\alpha}}^{\circ})}{\partial \alpha_j^{\circ}} + \kappa_0 h(\bar{\boldsymbol{\alpha}}^{\bullet}, \bar{\boldsymbol{\alpha}}^{\circ}) \frac{\partial h(\bar{\boldsymbol{\alpha}}^{\bullet}, \bar{\boldsymbol{\alpha}}^{\circ})}{\partial \alpha_j^{\circ}} + \frac{1}{\kappa_1} \frac{\partial B_1(\bar{\boldsymbol{\alpha}}^{\bullet}, \bar{\boldsymbol{\alpha}}^{\circ})}{\partial \alpha_j^{\circ}} + \frac{1}{\kappa_2} \frac{\partial B_2(\bar{\boldsymbol{\alpha}}^{\bullet}, \bar{\boldsymbol{\alpha}}^{\circ})}{\partial \alpha_j^{\circ}} \right) + \nu_{\alpha_j^{\circ}}(t), \quad (4.34)$$

for $\alpha_j(0) = \alpha_j^{(0)}$.

Equation 4.33 and 4.34 are finally rewritten as discrete-time equivalents

$$\alpha_j^{\bullet}[n+1] = \alpha_j^{\bullet}[n] - \mu_j \left[ \phi_{\alpha_j^{\bullet}}(\bar{\boldsymbol{\alpha}}^{\bullet}[n], \bar{\boldsymbol{\alpha}}^{\circ}[n]) + \kappa_0 h(\bar{\boldsymbol{\alpha}}^{\bullet}[n], \bar{\boldsymbol{\alpha}}^{\circ}[n]) + \frac{1}{\kappa_1} B_{1\,\alpha_j^{\bullet}}(\alpha_j^{\bullet}[n], \alpha_j^{\circ}[n]) + \frac{1}{\kappa_2} B_{2\,\alpha_j^{\bullet}}(\alpha_j^{\bullet}[n], \alpha_j^{\circ}[n]) \right] + \nu_{\alpha_j^{\bullet}}[n], \quad (4.35)$$

and

$$\alpha_j^{\circ}[n+1] = \alpha_j^{\circ}[n] - \rho_j \left[ \phi_{\alpha_j^{\circ}}(\bar{\boldsymbol{\alpha}}^{\bullet}[n], \bar{\boldsymbol{\alpha}}^{\circ}[n]) - \kappa_0 h(\bar{\boldsymbol{\alpha}}^{\bullet}[n], \bar{\boldsymbol{\alpha}}^{\circ}[n]) + \frac{1}{\kappa_1} B_{1\,\alpha_j^{\circ}}(\alpha_j^{\bullet}[n], \alpha_j^{\circ}[n]) + \frac{1}{\kappa_2} B_{2\,\alpha_j^{\circ}}(\alpha_j^{\bullet}[n], \alpha_j^{\circ}[n]) \right] + \nu_{\alpha_j^{\circ}}[n], \quad (4.36)$$

for $\alpha_j[0] = \alpha_j^{[0]}$, where

$$\phi_{\alpha_j^{\bullet}}(\bar{\boldsymbol{\alpha}}^{\bullet}[n], \bar{\boldsymbol{\alpha}}^{\circ}[n]) = \sum_{i=1}^{k} (\alpha_i^{\bullet} - \alpha_i^{\circ}) \psi(\bar{\boldsymbol{x}}_i, \bar{\boldsymbol{x}}_j) - y_j + \eta, \quad \text{and} \quad (4.37)$$

$$\phi_{\alpha_j^{\circ}}(\bar{\boldsymbol{\alpha}}^{\bullet}[n], \bar{\boldsymbol{\alpha}}^{\circ}[n]) = \sum_{i=1}^{k} (\alpha_i^{\circ} - \alpha_i^{\bullet}) \psi(\bar{\boldsymbol{x}}_i, \bar{\boldsymbol{x}}_j) + y_j + \eta, \quad (4.38)$$

where $m = 1, \cdots, k$, and

$$B_{1\,\alpha_j^{\bullet}}(\alpha_j^{\bullet}[n], \alpha_j^{\circ}[n]) = 0, \quad (4.39)$$

$$B_{2\,\alpha_j^{\bullet}}(\alpha_j^{\bullet}[n], \alpha_j^{\circ}[n]) = \begin{cases} \dfrac{1}{(C - \alpha_j^{\bullet})^2}, & \text{if} \quad C - \alpha_j^{\bullet} \geq \epsilon, \\ \dfrac{1}{\epsilon^2}, & \text{if} \quad C - \alpha_j^{\bullet} < \epsilon, \end{cases} \quad (4.40)$$

$$B_{1\,\alpha_j^{\circ}}(\alpha_j^{\bullet}[n], \alpha_j^{\circ}[n]) = \begin{cases} -\dfrac{1}{(\alpha_j^{\circ})^2}, & \text{if} \quad \alpha_j^{\circ} \geq \epsilon, \\ -\dfrac{1}{\epsilon^2}, & \text{if} \quad \alpha_j^{\circ} < \epsilon, \quad \text{and} \end{cases} \quad (4.41)$$

$$B_{2\,\alpha_j^{\circ}}(\alpha_j^{\bullet}[n], \alpha_j^{\circ}[n]) = 0. \quad (4.42)$$

Figure 4.11 and Fig. 4.12 illustrate implementable functional block diagrams of the ANNs

shown in Eq. 4.35 and Eq. 4.36 for one optimisation variable $\alpha_j^{\bullet}$ and $\alpha_j^{\circ}$ respectively; for diagram clarity only optimisation variable inputs are shown.

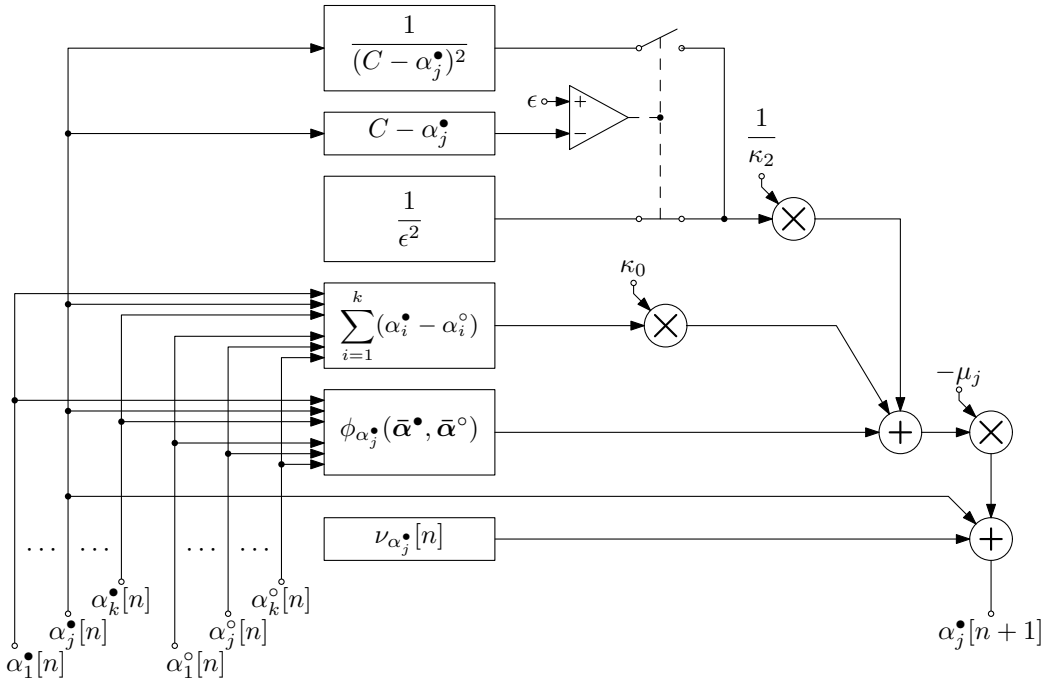



***Figure 4.11:*** *Functional block-diagram of the* ANN*-mapped combined exterior penalty function and interior penalty / barrier function optimisation technique for* SVM *regression as defined in Eq. 4.35.*

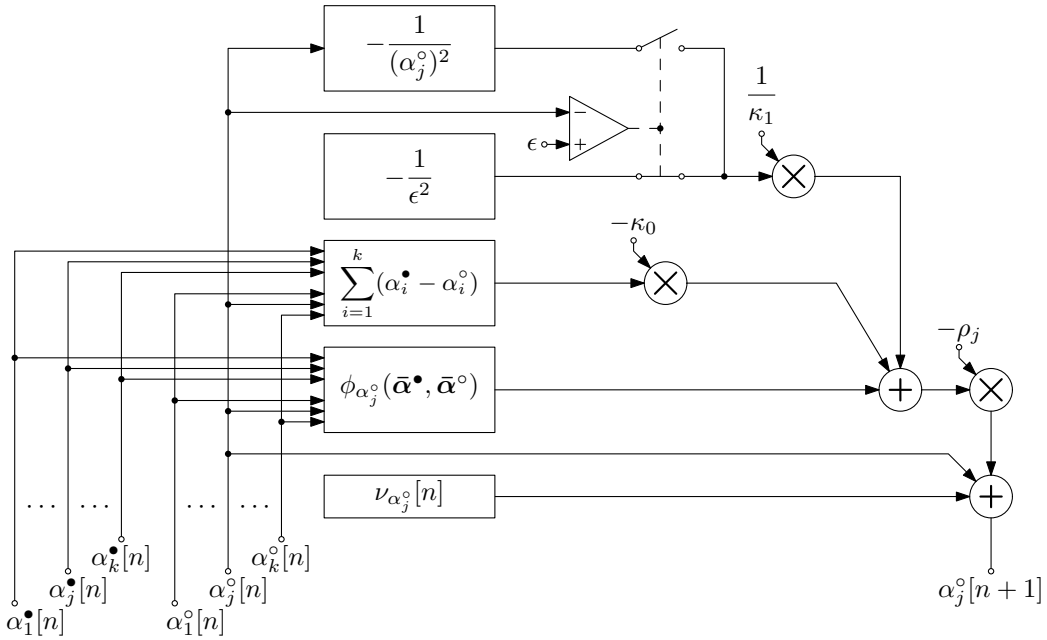



***Figure 4.12:*** *Functional block-diagram of the* ANN*-mapped combined exterior penalty function and interior penalty / barrier function optimisation technique for* SVM *regression as defined in Eq. 4.36.*

### 4.1.3 Augmented Lagrange Multiplier Optimisation

The Augmented Lagrange Multiplier ANN optimisation technique, shown in Eq. 3.72 and Eq. 3.73, was developed for the SVM classification problem as follows. The soft-margin support vector model is trained by optimising the Lagrangian

$$\arg\max_{\bar{\boldsymbol{\alpha}}} \phi'(\bar{\boldsymbol{\alpha}}) = \arg\max_{\bar{\boldsymbol{\alpha}}} \left( \sum_{i=1}^{k} \alpha_i - \frac{1}{2} \sum_{i=1}^{k} \sum_{j=1}^{k} \alpha_i \alpha_j y_i y_j \psi(\bar{\boldsymbol{x}}_i, \bar{\boldsymbol{x}}_j) \right), \tag{4.43}$$

subject to the constraints

$$\sum_{i=1}^{k} \alpha_i y_i = 0, \tag{4.44}$$

$$\alpha_i \geq 0, \text{ and} \tag{4.45}$$

$$C - \alpha_i \geq 0, \tag{4.46}$$

where $i = 1, \cdots, k$ and $C$ is the cost constant.

The augmented Lagrangian is constructed from the constrained optimisation problem shown in Eq. 4.43, Eq. 4.44, Eq. 4.45, and Eq. 4.45 and takes the form

$$\begin{aligned} L(\bar{\boldsymbol{\alpha}}, \bar{\boldsymbol{\lambda}}, \bar{\boldsymbol{\kappa}}) &= \phi(\bar{\boldsymbol{\alpha}}) + \lambda_0 h_0(\bar{\boldsymbol{\alpha}}) + \frac{\kappa_1}{2} h_0^2(\bar{\boldsymbol{\alpha}}) \\ &+ \sum_{j=1}^{k} \left( \lambda_j g_j(\alpha_j) + \frac{\kappa_2}{2} g_j^2(\alpha_j) \right) + \sum_{j=1}^{k} \left( \lambda_{k+j} g_{k+j}(\alpha_j) + \frac{\kappa_3}{2} g_{k+j}^2(\alpha_j) \right) \end{aligned} \tag{4.47}$$

where

$$\phi(\bar{\boldsymbol{\alpha}}) = -\sum_{i=1}^{k} \alpha_i + \frac{1}{2} \sum_{i=1}^{k} \sum_{j=1}^{k} \alpha_i \alpha_j y_i y_j \psi(\bar{\boldsymbol{x}}_i, \bar{\boldsymbol{x}}_j), \tag{4.48}$$

$$h_0(\bar{\boldsymbol{\alpha}}) = \sum_{i=1}^{k} \alpha_i y_i, \tag{4.49}$$

and

$$g_j(\alpha_j) = \begin{cases} \alpha_j, & \text{if} \quad \alpha_j < -\dfrac{\lambda_j}{\kappa_j}, \\ -\dfrac{\lambda_j}{\kappa_j}, & \text{if} \quad \alpha_j \geq -\dfrac{\lambda_j}{\kappa_j}, \end{cases} \tag{4.50}$$

$$g_{k+j}(\alpha_j) = \begin{cases} C - \alpha_j, & \text{if} \quad C - \alpha_j < -\dfrac{\lambda_{k+j}}{\kappa_{k+j}}, \\ -\dfrac{\lambda_{k+j}}{\kappa_{k+j}}, & \text{if} \quad C - \alpha_j \geq -\dfrac{\lambda_{k+j}}{\kappa_{k+j}}, \end{cases} \tag{4.51}$$

where $j = 1, \cdots, n$ and $\kappa_i \geq 0$ is the penalty parameter. The minimisation of the augmented Lagrangian can be converted into a system of differential equations

$$\frac{\mathrm{d}\bar{\boldsymbol{\alpha}}}{\mathrm{d}t} = -\boldsymbol{\mu}\nabla_{\bar{\boldsymbol{\alpha}}} L(\bar{\boldsymbol{\alpha}}, \bar{\boldsymbol{\lambda}}, \bar{\boldsymbol{\kappa}}) + \bar{\boldsymbol{\nu}}_{\alpha}(t), \tag{4.52}$$

$$\frac{\mathrm{d}\bar{\boldsymbol{\lambda}}}{\mathrm{d}t} = \quad \boldsymbol{\rho}\nabla_{\bar{\boldsymbol{\lambda}}} L(\bar{\boldsymbol{\alpha}}, \bar{\boldsymbol{\lambda}}, \bar{\boldsymbol{\kappa}}) + \bar{\boldsymbol{\nu}}_{\lambda}(t), \tag{4.53}$$

where $\boldsymbol{\mu} = \mathrm{diag}(\mu_1, \cdots, \mu_n)$ and $\boldsymbol{\rho} = \mathrm{diag}(\rho_1, \cdots, \rho_m)$ are positive scalar variables, typically chosen as $\mu_i > 0$ and $\rho_i > 0$, and with the initial conditions $\bar{\boldsymbol{\alpha}}(0) = \bar{\boldsymbol{\alpha}}^{(0)}$ and $\bar{\boldsymbol{\lambda}}(0) = \bar{\boldsymbol{\lambda}}^{(0)}$. This can be written in scalar form

$$\begin{aligned}\frac{\mathrm{d}\alpha_j}{\mathrm{d}t} = -\mu_j \Bigg( & \frac{\partial \phi(\bar{\boldsymbol{\alpha}})}{\partial \alpha_j} + \lambda_0 \frac{\partial h_0(\bar{\boldsymbol{\alpha}})}{\partial \alpha_j} + \frac{\kappa_0}{2} h_0(\bar{\boldsymbol{\alpha}}) \frac{\partial h_0(\bar{\boldsymbol{\alpha}})}{\partial \alpha_j} \\ & + \lambda_j \frac{\partial g_j(\alpha_j)}{\partial \alpha_j} + \frac{\kappa_j}{2} g_j(\alpha_j) \frac{\partial g_j(\alpha_j)}{\partial \alpha_j} + \\ & \lambda_{k+j} \frac{\partial g_{k+j}(\alpha_j)}{\partial \alpha_j} + \frac{\kappa_{k+j}}{2} g_{k+j}(\alpha_j) \frac{\partial g_{k+j}(\alpha_j)}{\partial \alpha_j} \Bigg) + \nu_{\alpha_j}(t),\end{aligned} \tag{4.54}$$

and

$$\frac{\mathrm{d}\lambda_0}{\mathrm{d}t} = \quad \rho_0\big(h_0(\bar{\boldsymbol{\alpha}})\big) + \nu_{\lambda_0}(t), \tag{4.55}$$

$$\frac{\mathrm{d}\lambda_j}{\mathrm{d}t} = \begin{cases} \rho_j\big(\alpha_j\big) + \nu_{\lambda_j}(t), & \text{if} \quad \alpha_j < -\dfrac{\lambda_j}{\kappa_j}, \\ \nu_{\lambda_j}(t), & \text{if} \quad \alpha_j \geq -\dfrac{\lambda_j}{\kappa_j}, \end{cases} \tag{4.56}$$

$$\frac{\mathrm{d}\lambda_{k+j}}{\mathrm{d}t} = \begin{cases} \rho_{k+j}\big(C - \alpha_j\big) + \nu_{\lambda_{k+j}}(t), & \text{if} \quad C - \alpha_j < -\dfrac{\lambda_{k+j}}{\kappa_{k+j}}, \\ \nu_{\lambda_{k+j}}(t), & \text{if} \quad C - \alpha_j \geq -\dfrac{\lambda_{k+j}}{\kappa_{k+j}}, \end{cases} \tag{4.57}$$

Equation 4.54, Eq. 4.55, Eq. 4.56, and Eq. 4.57 are finally rewritten as discrete-time equivalents

$$\begin{aligned}\alpha_j[n+1] = \alpha_j[n] - \mu_j \Bigg[ & \phi_{\alpha_j}(\bar{\boldsymbol{\alpha}}) + h_{0\alpha_j}(\bar{\boldsymbol{\alpha}}) \Big( \lambda_0[n] + \frac{\kappa_0}{2} h_0(\bar{\boldsymbol{\alpha}}) \Big) \\ & + g_{j\alpha_j}(\alpha_j) \Big( \lambda_j[n] + \frac{\kappa_j}{2} g_j(\alpha_j) \Big) \\ & + g_{k+j\alpha_j}(\alpha_j) \Big( \lambda_{k+j}[n] + \frac{\kappa_{k+j}}{2} g_{k+j}(\alpha_j) \Big) \Bigg] + \nu_{\alpha_j}[n],\end{aligned} \tag{4.58}$$

and

$$\lambda_0[n+1] = \lambda_0[n] + \rho_0\Big[h_0(\bar{\boldsymbol{\alpha}})\Big] + \nu_{\lambda_0}[n], \tag{4.59}$$

$$\lambda_j[n+1] = \begin{cases} \lambda_j[n] + \rho_j\Big[\alpha_j[n]\Big] + \nu_{\lambda_j}[n], & \text{if} \quad \alpha_j[n] < -\dfrac{\lambda_j[n]}{\kappa_j}, \\ \lambda_j[n] + \nu_{\lambda_j}[n], & \text{if} \quad \alpha_j[n] \geq -\dfrac{\lambda_j[n]}{\kappa_j}, \end{cases} \tag{4.60}$$

$$\lambda_{k+j}[n+1] = \begin{cases} \lambda_{k+j}[n] + \rho_{k+j}\Big[C - \alpha_j[n]\Big] + \nu_{\lambda_{k+j}}[n], & \text{if} \quad C - \alpha_j[n] < -\dfrac{\lambda_{k+j}[n]}{\kappa_{k+j}}, \\ \lambda_{k+j}[n] + \nu_{\lambda_{k+j}}[n], & \text{if} \quad C - \alpha_j[n] \geq -\dfrac{\lambda_{k+j}[n]}{\kappa_{k+j}}, \end{cases} \tag{4.61}$$

where

$$\phi_{\alpha_j}(\bar{\boldsymbol{\alpha}}) = \frac{1}{2}\sum_{i=1}^{k} \alpha_i[n] y_i y_j \psi(\bar{\boldsymbol{x}}_i, \bar{\boldsymbol{x}}_j) - 1, \tag{4.62}$$

$$h_0(\bar{\boldsymbol{\alpha}}) = \sum_{i=1}^{k} \alpha_i[n] y_i, \tag{4.63}$$

$$h_{0\alpha_j}(\bar{\boldsymbol{\alpha}}) = y_j, \tag{4.64}$$

$$g_j(\alpha_j) = \begin{cases} \alpha_j[n], & \text{if} \quad \alpha_j[n] < -\dfrac{\lambda_j[n]}{\kappa_j}, \\ -\dfrac{\lambda_j[n]}{\kappa_j}, & \text{if} \quad \alpha_j[n] \geq -\dfrac{\lambda_j[n]}{\kappa_j}, \end{cases} \tag{4.65}$$

$$g_{j\alpha_j}(\alpha_j) = \begin{cases} 1, & \text{if} \quad \alpha_j[n] < -\dfrac{\lambda_j[n]}{\kappa_j}, \\ 0, & \text{if} \quad \alpha_j[n] \geq -\dfrac{\lambda_j[n]}{\kappa_j}, \end{cases} \tag{4.66}$$

$$g_{k+j}(\alpha_j) = \begin{cases} C - \alpha_j[n], & \text{if} \quad C - \alpha_j[n] < -\dfrac{\lambda_{k+j}[n]}{\kappa_{k+j}}, \\ -\dfrac{\lambda_{k+j}[n]}{\kappa_{k+j}}, & \text{if} \quad C - \alpha_j[n] \geq -\dfrac{\lambda_{k+j}[n]}{\kappa_{k+j}}, \end{cases} \tag{4.67}$$

$$g_{k+j\alpha_j}(\alpha_j) = \begin{cases} -1, & \text{if} \quad C - \alpha_j[n] < -\dfrac{\lambda_{k+j}[n]}{\kappa_{k+j}}, \\ 0, & \text{if} \quad C - \alpha_j[n] \geq -\dfrac{\lambda_{k+j}[n]}{\kappa_{k+j}}, \end{cases} \tag{4.68}$$

Figure 4.13 and Fig. 4.14 illustrate implementable functional block diagrams of the ANN shown in Eq. 4.58, Eq. 4.59, Eq. 4.60 and Eq. 4.61 for the optimisation variables $\alpha_j, \lambda_0, \lambda_j$ and $\lambda_{k+j}$; for diagram clarity only optimisation variable inputs are shown.

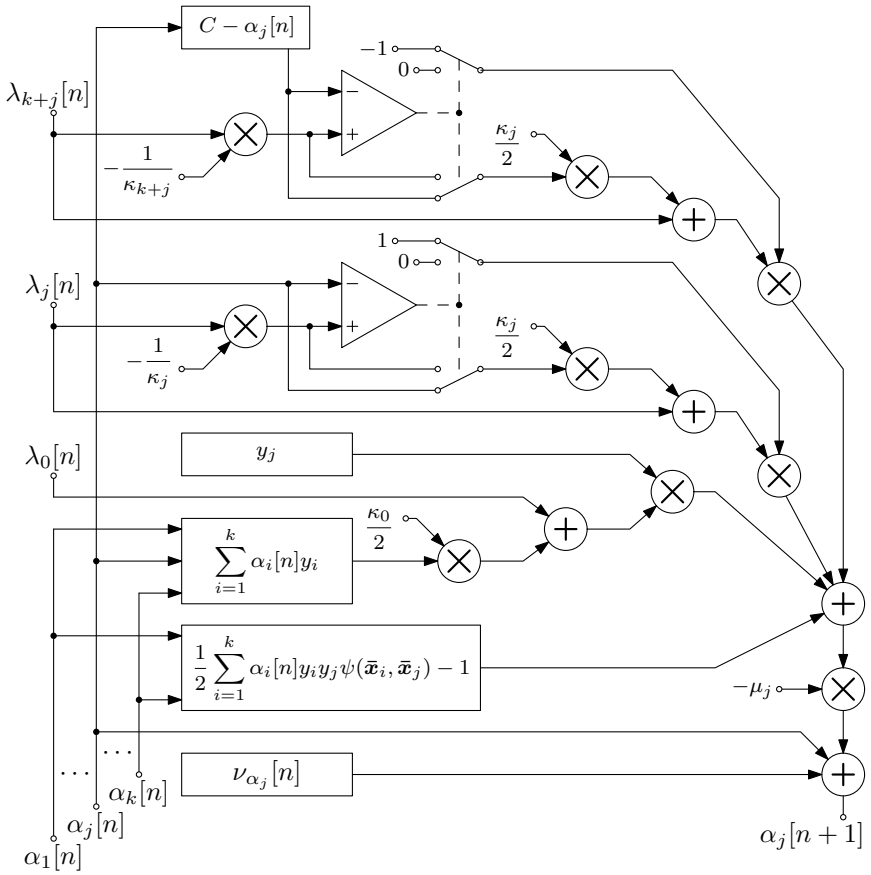



***Figure 4.13:*** *Functional block-diagram of the* ANN*-mapped augmented Lagrange Multiplier optimisation technique as defined in Eq. 4.58.*

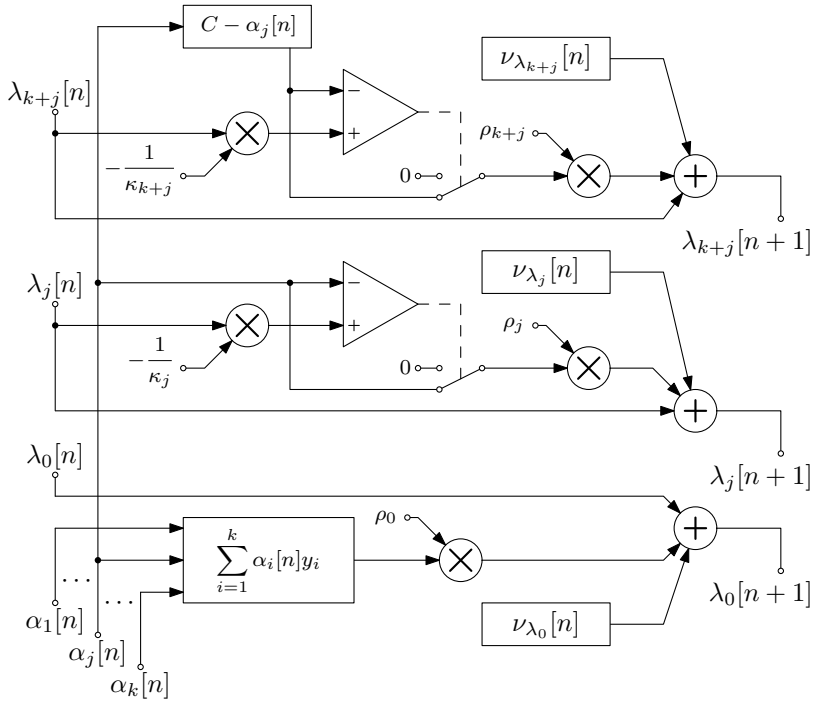



***Figure 4.14:*** *Functional block-diagram of the* ANN*-mapped augmented Lagrange Multiplier optimisation technique as defined in Eq. 4.59, Eq. 4.60, and Eq. 4.61.*

The Augmented Lagrange Multiplier ANN optimisation technique, shown in Eq. 3.72 and Eq. 3.73, was developed for the SVM regression problem as follows. The soft-margin support vector model for regression is trained by optimising the Lagrangian

$$\arg\max_{\bar{\boldsymbol{\alpha}}^{\bullet},\bar{\boldsymbol{\alpha}}^{\circ}} \phi'(\bar{\boldsymbol{\alpha}}^{\bullet},\bar{\boldsymbol{\alpha}}^{\circ}) = \arg\max_{\bar{\boldsymbol{\alpha}}^{\bullet},\bar{\boldsymbol{\alpha}}^{\circ}} \left( -\frac{1}{2}\sum_{i=1}^{k}\sum_{j=1}^{k}(\alpha_i^{\bullet}-\alpha_i^{\circ})(\alpha_j^{\bullet}-\alpha_j^{\circ})\psi(\bar{\boldsymbol{x}}_i,\bar{\boldsymbol{x}}_j) + \sum_{i=1}^{k} y_i(\alpha_i^{\bullet}-\alpha_i^{\circ}) - \eta\sum_{i=1}^{k}(\alpha_i^{\bullet}+\alpha_i^{\circ}) \right), \quad (4.69)$$

subject to the constraints

$$\sum_{i=1}^{k}(\alpha_i^{\bullet}-\alpha_i^{\circ}) = 0, \quad (4.70)$$

$$\alpha_i^{\circ} \geq 0, \text{ and} \quad (4.71)$$

$$C-\alpha_i^{\bullet} \geq 0, \quad (4.72)$$

where $i = 1, \cdots, k$ and $C$ is the cost constant.

The augmented Lagrangian is constructed from the constrained optimisation problem shown in Eq. 4.69, Eq. 4.70, Eq. 4.71, and Eq. 4.71 and takes the form

$$L(\bar{\boldsymbol{\alpha}}^{\bullet},\bar{\boldsymbol{\alpha}}^{\circ},\bar{\boldsymbol{\lambda}},\bar{\boldsymbol{\kappa}}) = \phi(\bar{\boldsymbol{\alpha}}^{\bullet},\bar{\boldsymbol{\alpha}}^{\circ}) + \lambda_0 h_0(\bar{\boldsymbol{\alpha}}^{\bullet},\bar{\boldsymbol{\alpha}}^{\circ}) + \frac{\kappa_1}{2}h_0^2(\bar{\boldsymbol{\alpha}}^{\bullet},\bar{\boldsymbol{\alpha}}^{\circ}) + \sum_{j=1}^{k}\left(\lambda_j g_j(\alpha_j^{\bullet},\alpha_j^{\circ}) + \frac{\kappa_2}{2}g_j^2(\alpha_j^{\bullet},\alpha_j^{\circ})\right) + \sum_{j=1}^{k}\left(\lambda_{k+j} g_{k+j}(\alpha_j^{\bullet},\alpha_j^{\circ}) + \frac{\kappa_3}{2}g_{k+j}^2(\alpha_j^{\bullet},\alpha_j^{\circ})\right) \quad (4.73)$$

where

$$\phi(\bar{\boldsymbol{\alpha}}^{\bullet},\bar{\boldsymbol{\alpha}}^{\circ}) = \frac{1}{2}\sum_{i=1}^{k}\sum_{j=1}^{k}(\alpha_i^{\bullet}-\alpha_i^{\circ})(\alpha_j^{\bullet}-\alpha_j^{\circ})\psi(\bar{\boldsymbol{x}}_i,\bar{\boldsymbol{x}}_j) - \sum_{i=1}^{k} y_i(\alpha_i^{\bullet}-\alpha_i^{\circ}) + \eta\sum_{i=1}^{k}(\alpha_i^{\bullet}+\alpha_i^{\circ}), \quad (4.74)$$

$$h_0(\bar{\boldsymbol{\alpha}}^{\bullet},\bar{\boldsymbol{\alpha}}^{\circ}) = \sum_{i=1}^{k}(\alpha_i^{\bullet}-\alpha_i^{\circ}), \quad (4.75)$$

and

$$g_j(\alpha_j^\bullet, \alpha_j^\circ) = \begin{cases} \alpha_j^\circ, & \text{if} \quad \alpha_j^\circ < -\dfrac{\lambda_j}{\kappa_j}, \\ -\dfrac{\lambda_j}{\kappa_j}, & \text{if} \quad \alpha_j^\circ \geq -\dfrac{\lambda_j}{\kappa_j}, \end{cases} \tag{4.76}$$

$$g_{k+j}(\alpha_j^\bullet, \alpha_j^\circ) = \begin{cases} C - \alpha_j^\bullet, & \text{if} \quad C - \alpha_j^\bullet < -\dfrac{\lambda_{k+j}}{\kappa_{k+j}}, \\ -\dfrac{\lambda_{k+j}}{\kappa_{k+j}}, & \text{if} \quad C - \alpha_j^\bullet \geq -\dfrac{\lambda_{k+j}}{\kappa_{k+j}}, \end{cases} \tag{4.77}$$

where $j = 1, \cdots, n$ and $\kappa_i \geq 0$ is the penalty parameter. The minimisation of the augmented Lagrangian can be converted into a system of differential equations

$$\frac{\mathrm{d}\bar{\boldsymbol{\alpha}}^\bullet}{\mathrm{d}t} = -\boldsymbol{\mu}_{\bar{\boldsymbol{\alpha}}^\bullet} \nabla_{\bar{\boldsymbol{\alpha}}^\bullet} L(\bar{\boldsymbol{\alpha}}^\bullet, \bar{\boldsymbol{\alpha}}^\circ, \bar{\boldsymbol{\lambda}}, \bar{\boldsymbol{\kappa}}) + \bar{\boldsymbol{\nu}}_{\alpha^\bullet}(t), \tag{4.78}$$

$$\frac{\mathrm{d}\bar{\boldsymbol{\alpha}}^\circ}{\mathrm{d}t} = -\boldsymbol{\mu}_{\bar{\boldsymbol{\alpha}}^\circ} \nabla_{\bar{\boldsymbol{\alpha}}^\circ} L(\bar{\boldsymbol{\alpha}}^\circ, \bar{\boldsymbol{\alpha}}^\circ, \bar{\boldsymbol{\lambda}}, \bar{\boldsymbol{\kappa}}) + \bar{\boldsymbol{\nu}}_{\alpha^\circ}(t), \tag{4.79}$$

$$\frac{\mathrm{d}\bar{\boldsymbol{\lambda}}}{\mathrm{d}t} = \quad \boldsymbol{\rho} \nabla_{\bar{\boldsymbol{\lambda}}} L(\bar{\boldsymbol{\alpha}}^\bullet, \bar{\boldsymbol{\alpha}}^\circ, \bar{\boldsymbol{\lambda}}, \bar{\boldsymbol{\kappa}}) + \bar{\boldsymbol{\nu}}_{\lambda}(t), \tag{4.80}$$

where $\boldsymbol{\mu} = \mathrm{diag}(\mu_1, \cdots, \mu_n)$ and $\boldsymbol{\rho} = \mathrm{diag}(\rho_1, \cdots, \rho_m)$ are positive scalar variables, typically chosen as $\mu_i > 0$ and $\rho_i > 0$, and with the initial conditions $\bar{\boldsymbol{\alpha}}(0) = \bar{\boldsymbol{\alpha}}^{(0)}$ and $\bar{\boldsymbol{\lambda}}(0) = \bar{\boldsymbol{\lambda}}^{(0)}$. This can be written in scalar form

$$\begin{aligned} \frac{\mathrm{d}\alpha_j^\bullet}{\mathrm{d}t} = -\mu_{\alpha_j^\bullet} \Bigg( & \frac{\partial \phi(\bar{\boldsymbol{\alpha}}^\bullet, \bar{\boldsymbol{\alpha}}^\circ)}{\partial \alpha_j^\bullet} + \lambda_0 \frac{\partial h_0(\bar{\boldsymbol{\alpha}}^\bullet, \bar{\boldsymbol{\alpha}}^\circ)}{\partial \alpha_j^\bullet} + \frac{\kappa_0}{2} h_0(\bar{\boldsymbol{\alpha}}^\bullet, \bar{\boldsymbol{\alpha}}^\circ) \frac{\partial h_0(\bar{\boldsymbol{\alpha}}^\bullet, \bar{\boldsymbol{\alpha}}^\circ)}{\partial \alpha_j^\bullet} \\ & + \lambda_j \frac{\partial g_j(\alpha_j^\bullet, \alpha_j^\circ)}{\partial \alpha_j^\bullet} + \frac{\kappa_j}{2} g_j(\alpha_j^\bullet, \alpha_j^\circ) \frac{\partial g_j(\alpha_j^\bullet, \alpha_j^\circ)}{\partial \alpha_j^\bullet} + \\ & \lambda_{k+j} \frac{\partial g_{k+j}(\alpha_j^\bullet, \alpha_j^\circ)}{\partial \alpha_j^\bullet} + \frac{\kappa_{k+j}}{2} g_{k+j}(\alpha_j^\bullet, \alpha_j^\circ) \frac{\partial g_{k+j}(\alpha_j^\bullet, \alpha_j^\circ)}{\partial \alpha_j^\bullet} \Bigg) + \nu_{\alpha_j^\bullet}(t), \end{aligned} \tag{4.81}$$

$$\begin{aligned} \frac{\mathrm{d}\alpha_j^\circ}{\mathrm{d}t} = -\mu_{\alpha_j^\circ} \Bigg( & \frac{\partial \phi(\bar{\boldsymbol{\alpha}}^\bullet, \bar{\boldsymbol{\alpha}}^\circ)}{\partial \alpha_j^\circ} + \lambda_0 \frac{\partial h_0(\bar{\boldsymbol{\alpha}}^\bullet, \bar{\boldsymbol{\alpha}}^\circ)}{\partial \alpha_j^\circ} + \frac{\kappa_0}{2} h_0(\bar{\boldsymbol{\alpha}}^\bullet, \bar{\boldsymbol{\alpha}}^\circ) \frac{\partial h_0(\bar{\boldsymbol{\alpha}}^\bullet, \bar{\boldsymbol{\alpha}}^\circ)}{\partial \alpha_j^\circ} \\ & + \lambda_j \frac{\partial g_j(\alpha_j^\bullet, \alpha_j^\circ)}{\partial \alpha_j^\circ} + \frac{\kappa_j}{2} g_j(\alpha_j^\bullet, \alpha_j^\circ) \frac{\partial g_j(\alpha_j^\bullet, \alpha_j^\circ)}{\partial \alpha_j^\circ} + \\ & \lambda_{k+j} \frac{\partial g_{k+j}(\alpha_j^\bullet, \alpha_j^\circ)}{\partial \alpha_j^\circ} + \frac{\kappa_{k+j}}{2} g_{k+j}(\alpha_j^\bullet, \alpha_j^\circ) \frac{\partial g_{k+j}(\alpha_j^\bullet, \alpha_j^\circ)}{\partial \alpha_j^\circ} \Bigg) + \nu_{\alpha_j^\circ}(t), \end{aligned} \tag{4.82}$$

and

$$\frac{\mathrm{d}\lambda_0}{\mathrm{d}t} = \quad \rho_0\big(h_0(\bar{\boldsymbol{\alpha}}^\bullet, \bar{\boldsymbol{\alpha}}^\circ)\big) + \nu_{\lambda_0}(t), \tag{4.83}$$

$$\frac{\mathrm{d}\lambda_j}{\mathrm{d}t} = \begin{cases} \rho_j\big(\alpha_j^\circ\big) + \nu_{\lambda_j}(t), & \text{if} \quad \alpha_j^\circ < -\dfrac{\lambda_j}{\kappa_j}, \\ \nu_{\lambda_j}(t), & \text{if} \quad \alpha_j^\circ \geq -\dfrac{\lambda_j}{\kappa_j}, \end{cases} \tag{4.84}$$

$$\frac{\mathrm{d}\lambda_{k+j}}{\mathrm{d}t} = \begin{cases} \rho_{k+j}\big(C - \alpha_j^\bullet\big) + \nu_{\lambda_{k+j}}(t), & \text{if} \quad C - \alpha_j^\bullet < -\dfrac{\lambda_{k+j}}{\kappa_{k+j}}, \\ \nu_{\lambda_{k+j}}(t), & \text{if} \quad C - \alpha_j^\bullet \geq -\dfrac{\lambda_{k+j}}{\kappa_{k+j}}, \end{cases} \tag{4.85}$$

Equation 4.81, Eq. 4.82 Eq. 4.83, Eq. 4.84, and Eq. 4.85 are finally rewritten as discrete-time equivalents

$$\begin{aligned} \alpha_j^\bullet[n+1] = \alpha_j^\bullet[n] - \mu_{\alpha_j^\bullet}\Bigg[ & \phi_{\alpha_j^\bullet}(\bar{\boldsymbol{\alpha}}^\bullet, \bar{\boldsymbol{\alpha}}^\circ) + h_{0\alpha_j^\bullet}(\bar{\boldsymbol{\alpha}}^\bullet, \bar{\boldsymbol{\alpha}}^\circ)\bigg(\lambda_0[n] + \frac{\kappa_0}{2} h_0(\bar{\boldsymbol{\alpha}}^\bullet, \bar{\boldsymbol{\alpha}}^\circ)\bigg) \\ & + g_{j\alpha_j^\bullet}(\alpha_j^\bullet, \alpha_j^\circ)\bigg(\lambda_j[n] + \frac{\kappa_j}{2} g_j(\alpha_j^\bullet, \alpha_j^\circ)\bigg) \\ & + g_{k+j\alpha_j^\bullet}(\alpha_j^\bullet, \alpha_j^\circ)\bigg(\lambda_{k+j}[n] + \frac{\kappa_{k+j}}{2} g_{k+j}(\alpha_j^\bullet, \alpha_j^\circ)\bigg)\Bigg] + \nu_{\alpha_j^\bullet}[n], \end{aligned} \tag{4.86}$$

$$\begin{aligned} \alpha_j^\circ[n+1] = \alpha_j^\circ[n] - \mu_{\alpha_j^\circ}\Bigg[ & \phi_{\alpha_j^\circ}(\bar{\boldsymbol{\alpha}}^\bullet, \bar{\boldsymbol{\alpha}}^\circ) + h_{0\alpha_j^\circ}(\bar{\boldsymbol{\alpha}}^\bullet, \bar{\boldsymbol{\alpha}}^\circ)\bigg(\lambda_0[n] + \frac{\kappa_0}{2} h_0(\bar{\boldsymbol{\alpha}}^\bullet, \bar{\boldsymbol{\alpha}}^\circ)\bigg) \\ & + g_{j\alpha_j^\circ}(\alpha_j^\bullet, \alpha_j^\circ)\bigg(\lambda_j[n] + \frac{\kappa_j}{2} g_j(\alpha_j^\bullet, \alpha_j^\circ)\bigg) \\ & + g_{k+j\alpha_j^\circ}(\alpha_j^\bullet, \alpha_j^\circ)\bigg(\lambda_{k+j}[n] + \frac{\kappa_{k+j}}{2} g_{k+j}(\alpha_j^\bullet, \alpha_j^\circ)\bigg)\Bigg] + \nu_{\alpha_j^\circ}[n], \end{aligned} \tag{4.87}$$

and

$$\lambda_0[n+1] = \lambda_0[n] + \rho_0\Big[h_0(\bar{\boldsymbol{\alpha}}^\bullet, \bar{\boldsymbol{\alpha}}^\circ)\Big] + \nu_{\lambda_0}[n], \tag{4.88}$$

$$\lambda_j[n+1] = \begin{cases} \lambda_j[n] + \rho_j\Big[\alpha_j^\circ[n]\Big] + \nu_{\lambda_j}[n], & \text{if} \quad \alpha_j^\circ[n] < -\dfrac{\lambda_j[n]}{\kappa_j}, \\ \lambda_j[n] + \nu_{\lambda_j}[n], & \text{if} \quad \alpha_j^\circ[n] \geq -\dfrac{\lambda_j[n]}{\kappa_j}, \end{cases} \tag{4.89}$$

$$\lambda_{k+j}[n+1] = \begin{cases} \lambda_{k+j}[n] + \rho_{k+j}\Big[C - \alpha_j^\bullet[n]\Big] + \nu_{\lambda_{k+j}}[n], & \text{if} \quad C - \alpha_j^\bullet[n] < -\dfrac{\lambda_{k+j}[n]}{\kappa_{k+j}}, \\ \lambda_{k+j}[n] + \nu_{\lambda_{k+j}}[n], & \text{if} \quad C - \alpha_j^\bullet[n] \geq -\dfrac{\lambda_{k+j}[n]}{\kappa_{k+j}}, \end{cases} \tag{4.90}$$

where

$$\phi_{\alpha_j^\bullet}(\bar{\boldsymbol{\alpha}}^\bullet[n], \bar{\boldsymbol{\alpha}}^\circ[n]) = \sum_{i=1}^{k}(\alpha_i^\bullet - \alpha_i^\circ)\psi(\bar{\boldsymbol{x}}_i, \bar{\boldsymbol{x}}_j) - y_j + \eta, \quad \text{and} \tag{4.91}$$

$$\phi_{\alpha_j^\circ}(\bar{\boldsymbol{\alpha}}^\bullet[n], \bar{\boldsymbol{\alpha}}^\circ[n]) = \sum_{i=1}^{k}(\alpha_i^\circ - \alpha_i^\bullet)\psi(\bar{\boldsymbol{x}}_i, \bar{\boldsymbol{x}}_j) + y_j + \eta, \tag{4.92}$$

$$h_0(\bar{\boldsymbol{\alpha}}^\bullet, \bar{\boldsymbol{\alpha}}^\circ) = \sum_{i=1}^{k}(\alpha_i^\bullet - \alpha_i^\circ), \tag{4.93}$$

$$h_{0\alpha_j^\bullet}(\bar{\boldsymbol{\alpha}}^\bullet, \bar{\boldsymbol{\alpha}}^\circ) = \quad 1, \tag{4.94}$$

$$h_{0\alpha_j^\circ}(\bar{\boldsymbol{\alpha}}^\bullet, \bar{\boldsymbol{\alpha}}^\circ) = -1, \tag{4.95}$$

$$g_j(\alpha_j^\bullet, \alpha_j^\circ) = \begin{cases} \alpha_j[n]^\circ, & \text{if} \quad \alpha_j[n]^\circ < -\dfrac{\lambda_j[n]}{\kappa_j}, \\ -\dfrac{\lambda_j[n]}{\kappa_j}, & \text{if} \quad \alpha_j[n]^\circ \geq -\dfrac{\lambda_j[n]}{\kappa_j}, \end{cases} \tag{4.96}$$

$$g_{j\alpha_j^\bullet}(\alpha_j^\bullet, \alpha_j^\circ) = \begin{cases} 0, & \text{if} \quad \alpha_j[n]^\circ < -\dfrac{\lambda_j[n]}{\kappa_j}, \\ 0, & \text{if} \quad \alpha_j[n]^\circ \geq -\dfrac{\lambda_j[n]}{\kappa_j}, \end{cases} \tag{4.97}$$

$$g_{j\alpha_j^\circ}(\alpha_j^\bullet, \alpha_j^\circ) = \begin{cases} 1, & \text{if} \quad \alpha_j[n]^\circ < -\dfrac{\lambda_j[n]}{\kappa_j}, \\ 0, & \text{if} \quad \alpha_j[n]^\circ \geq -\dfrac{\lambda_j[n]}{\kappa_j}, \end{cases} \tag{4.98}$$

$$g_{k+j}(\alpha_j^\bullet, \alpha_j^\circ) = \begin{cases} C - \alpha_j[n]^\bullet, & \text{if} \quad C - \alpha_j[n]^\bullet < -\dfrac{\lambda_{k+j}[n]}{\kappa_{k+j}}, \\ -\dfrac{\lambda_{k+j}[n]}{\kappa_{k+j}}, & \text{if} \quad C - \alpha_j[n]^\bullet \geq -\dfrac{\lambda_{k+j}[n]}{\kappa_{k+j}}, \end{cases} \tag{4.99}$$

$$g_{k+j\alpha_j^\bullet}(\alpha_j^\bullet, \alpha_j^\circ) = \begin{cases} -1, & \text{if} \quad C - \alpha_j[n]^\bullet < -\dfrac{\lambda_{k+j}[n]}{\kappa_{k+j}}, \\ 0, & \text{if} \quad C - \alpha_j[n]^\bullet \geq -\dfrac{\lambda_{k+j}[n]}{\kappa_{k+j}}, \end{cases} \tag{4.100}$$

$$g_{k+j\alpha_j^\circ}(\alpha_j^\bullet, \alpha_j^\circ) = \begin{cases} 0, & \text{if} \quad C - \alpha_j[n]^\bullet < -\dfrac{\lambda_{k+j}[n]}{\kappa_{k+j}}, \\ 0, & \text{if} \quad C - \alpha_j[n]^\bullet \geq -\dfrac{\lambda_{k+j}[n]}{\kappa_{k+j}}, \end{cases} \tag{4.101}$$

$$\tag{4.102}$$

Figure 4.15, Fig. 4.16 and Fig. 4.17 illustrate implementable functional block diagrams of the ANN shown in Eq. 4.86, Eq. 4.87, Eq. 4.88, Eq. 4.89 and Eq. 4.90 for the optimisation variables $\alpha_j^\bullet, \alpha_j^\circ, \lambda_0, \lambda_j$ and $\lambda_{k+j}$; for diagram clarity only optimisation variable inputs are shown.

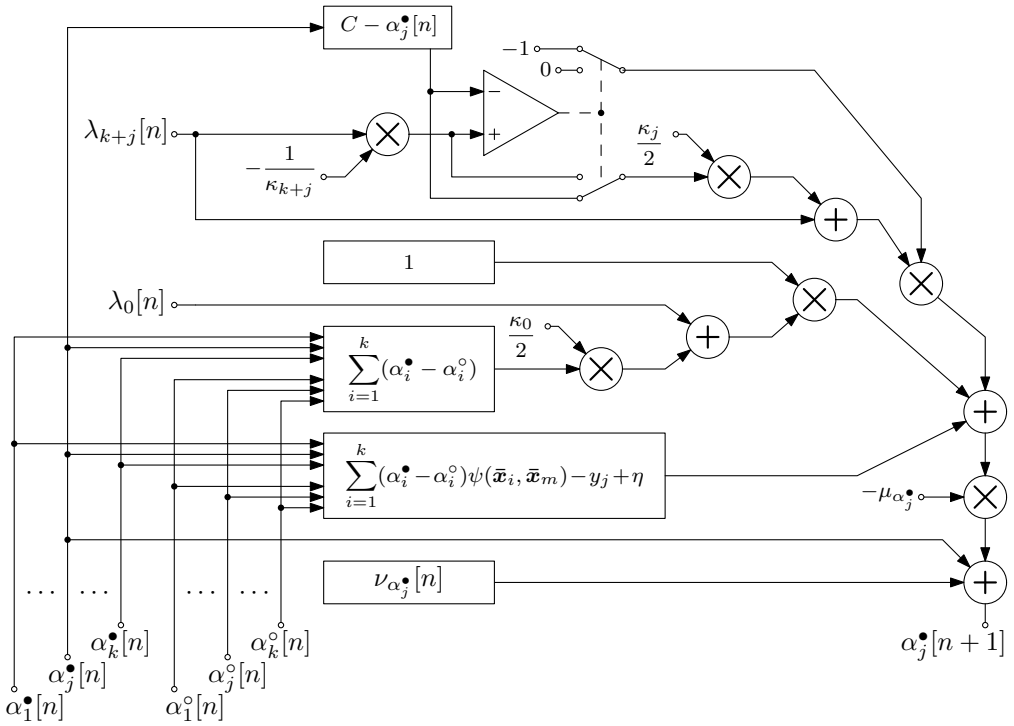



***Figure 4.15:*** *Functional block-diagram of the* ANN*-mapped augmented Lagrange Multiplier optimisation technique as defined in Eq. 4.86.*

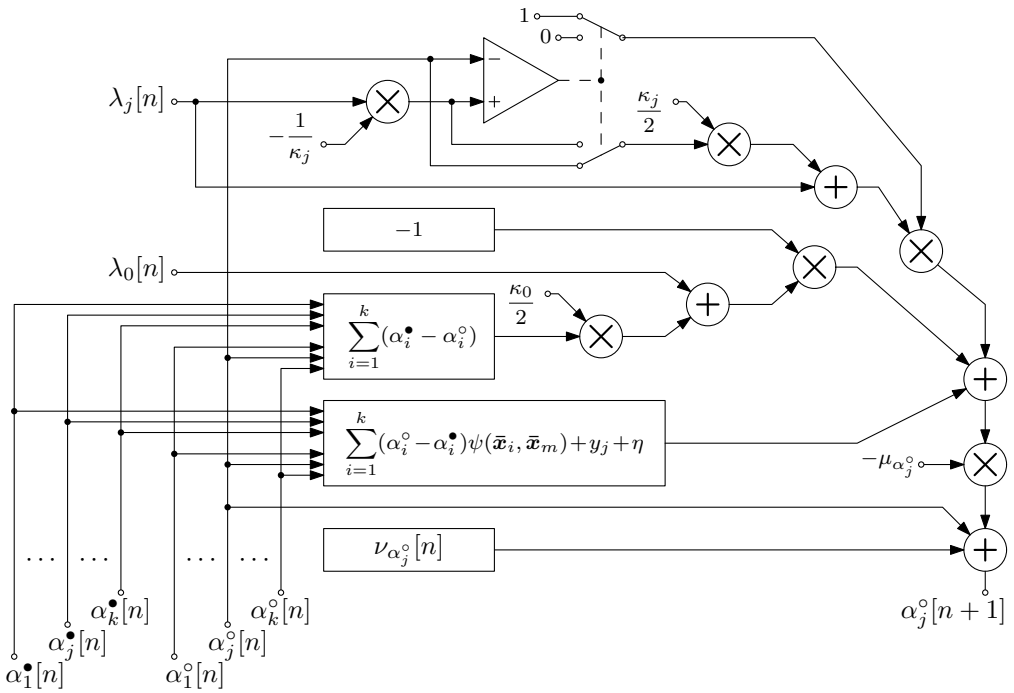



***Figure 4.16:*** *Functional block-diagram of the* ANN*-mapped augmented Lagrange Multiplier optimisation technique as defined in Eq. 4.87.*

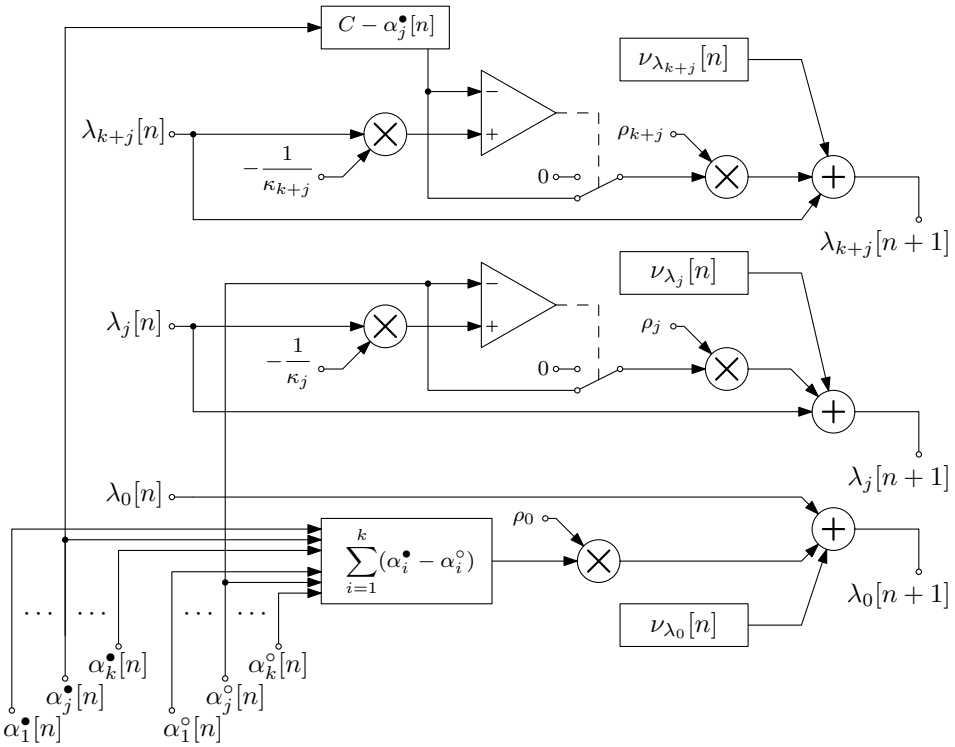



***Figure 4.17:*** *Functional block-diagram of the* ANN*-mapped augmented Lagrange Multiplier optimisation technique as defined in Eq. 4.88, Eq. 4.89, and Eq. 4.90.*

## 4.2 SVM Test-Rig System Hardware Architecture

This section provides details of the SVM test-rig system architecture design and implementation. This includes design methodology considerations, physical FPGA development platform considerations, FPGA development and supporting software considerations and procurement, and, system-level and modular low-level FPGA hardware architecture design and implementation details. As design methodology and hardware development practicalities are shared across both the SVM test-rig system hardware and the SVM DSP pipeline hardware all cursory DSP pipeline considerations are addressed in this subsection. However all specific technical design details are addressed in Section 4.3.

### 4.2.1 FPGA Platform and Implementation Considerations

The primary motivation for this research, and thus principal design performance parameter for the SVM system, is achieving real-time SVM performance - the computational latency or execution time is equal or is less than equal to the period between each successive input data sample. Therefore, the goal of this project is succinctly summarised by considering the following two research questions. *What is the minimum latency that can be achieved by an FPGA-based SVM implementation over some subset of design variations? What is the minimum period between each successive input data sample applied to an FPGA-based SVM implementation to maintain real-time operation?*

The core consideration and thus fundamental justification for implementing the SVM system as FPGA hardware is the goal of achieving real-time performance. Implemented via traditional software-based mechanisms, the maximum concurrently executed operations is limited by the number of cores a Central Processing Unit (CPU) is composed of, or, the number of threads each CPU core can concurrently execute. An FPGA hardware implementation by comparison can concurrently execute operations several orders of magnitude greater than a traditional CPU architecture. As the SVM is such a highly mathematical operation-rich paradigm it is thus well suited to a parallelised systolic pipelined implementation. Utilising a multi-threaded CPU based implementation such a system is still limited to some small finite-thread set of sequentially executed operations. An FPGA-based implementation can be designed such that massively-parallel and concurrent systolic pipelined operations are executed in large volume each system clock cycle.

Though an General-purpose Computing on Graphics Processing Units (GPGPU) based implementation could serve as a possible design solution, it would still require a PC-system backbone for operation. This is an undesirable solution and introduces unnecessary constraints and will pose infrastructural problems for field-based instrumentation adaptation.

An FPGA implementation allows for massively-parallel and systolic mathematical operation execution, and, concurrent pipelined execution of these operations. The decision to implement the system using FPGA technologies over other technologies was a matter

of sequential execution vs. systolic, parallel, and pipelined-processing-element execution, and the ease of such an implementation concerning both physical (standalone FPGA vs. GPGPU implementation and required supporting-infrastructure) and abstract implementation mechanisms (VHDL vs. software and API frameworks, such as CUDA and OpenCL).

Engineering parameters and metrics, such as gate-count, functional-block and DSP-block usage, and power consumption, are not presented as design constraints as they were secondary metrics to be measured and explored with design variation as part of this research work. This work was a pure research and development exercise as opposed to the development of an industrial or commercial product based on existing research and technological and financial constraints; these metrics were thus regarded as unknown during the design and development process. Minimising these metrics is, however, valid future work and will certainly feature amongst future product development requirements.

### 4.2.2 Design Methodology Considerations

The system was designed and implemented using both *concurrent development* and a combination of *top-down* and *bottom-up* modular design methodologies. The combination and application of these methodologies was appropriate for this research and development endeavour. By designing each modular subsystem implementation in parallel with an evolving system-wide design target and the greater development effort, inevitable design adjustments were easily made and ensured a final set of designs and implementations that were fit for final application purpose, and, provided appropriate information and metrics that contribute toward the goals of this research.

For example low-level modular FPGA processes such as multiplication and addition operations were implemented by various methods, were subsequently tested, and finally appropriate verification and functional testing outcomes were compared and contrasted. These low-level development tasks were completed concurrently with the design and development of the higher-level SVM system architectures. Once unexpected transient behaviour or erroneous operation were resolved or mitigated the verified subsystems in question were integrated, with minimal re-engineering or systems reintegration effort, to form higher-level subsystems that were again tested and verified accordingly. The higher-level modular subsystems were then again integrated until the ultimate-system design was achieved. At each subsystem integration stage any testing problems or hardware bugs that became apparent were easily identified and attributed to the offending subsystem implementation and appropriate modifications or mitigations were made. Through this design and development process subsystems were integrated as required and realised with minimal additional debugging and testing requirements.

A benefit observed through the application of concurrent top-down and bottom-up modular design and development methodologies in this work was the division of labour and

its consequent advantages. By concurrently maintaining several active research and development tasks, when any one of these pursuits stagnated or become obstructed for any reason, overall system development did not stop and thus research and development productivity was sustained. Also, the introduction of alternative techniques or methods acquired through new literature or development serendipity was quickly assimilated and integrated and into system designs by the introduction of either another modular component or trivial modification to an existing component.

All designs are, however, presented in a modular top-down manner in this thesis. This modular top-down presentation allows for a thorough design prescription while remaining both succinct and comprehensible to the reader and the author alike. A modular top-down presentation such as this also lends itself well to the comprehension of each modular VHDL implementation. Conversely the reproduction of complicated and unwieldy circuit-diagram designs, as observed in the bygone-era of discrete-logic design, is redundant due to the nature and application of the VHDL language and synthesis process. Such a pursuit requires the lengthy and error-prone partitioning of subsystems followed by their conversion back to high-level hardware descriptions compatible with modern FPGA development and synthesis tools. Such a divergence from modern digital-system design practises are universally deemed an inappropriate and inefficient endeavour, thus have no place in modern digital-system design [67] and do not feature in such a form in this thesis.

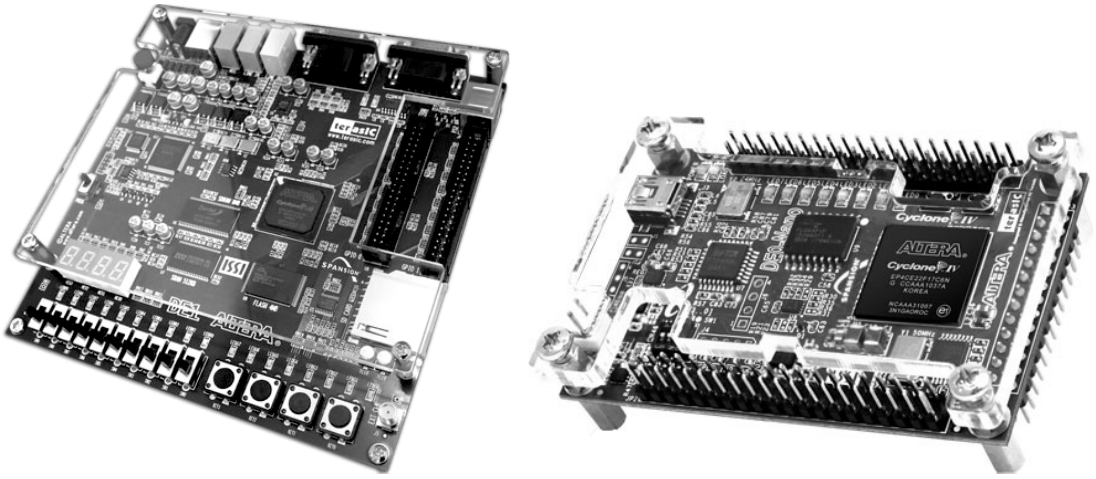
### 4.2.3 FPGA Development Platform Considerations

The selection of FPGA vendor, and thus target FPGA device, was made early in the work's design and development phases. The following exposition provides insight into nomination of Altera FPGA devices over their market competitor Xilinx FPGA devices. Factors that influenced these decisions included available development tools, access to FPGA development boards, and finally FPGA target device technology and technical specifications.

The author was familiar with both Xilinx and Altera development tools, software suites, and FPGA technology, and had developed digital logic targeting both vendor's respective FPGA technologies. Informed by this professional experience the author found Altera's development suite, Quartus II (now known as Quartus Prime [87]), superior to Xilinx's development suite ISE (now superseded by Xilinx Vivado Design Suite [88]) in ease of use, system compilation and synthesis error reporting, and system instantiation and FPGA programming.

Stemming from convenience and pragmatism the choice of FPGA vendor was also influenced by the development tools available to the author from the work's outset. The author was in possession of two Altera FPGA development boards; the Terasic *DE1* development board utilising an Altera Cyclone II FPGA, and, the Terasic *DE0-Nano* development board utilising an Altera Cyclone IV FPGA. From technology-generation,

hardware specification, and feature-set perspectives - including integrated USB JTAG programming and debugging features - the Altera-based devices outclassed the Xilinx Sparton-3 development boards available in the Department of Electrical and Electronic Engineering at the University of Nottingham. Thus initial design and development efforts began utilising the Terasic DE1 and DE0-Nano development boards. The Terasic DE1 and DE0-Nano development boards are shown in Fig. 4.18.



**(a)** **(b)**

***Figure 4.18:*** *Terasic Altera FPGA development boards; (a) the DE1 Cyclone II development board, and (b) the DE0-Nano Cyclone IV development board.*

Research funding was awarded in the form of a PRGS grant from the Government of Malaysia. The grant's focus was the development of a prototype real-time oil and gas pipeline defect and failure detection system utilising the real-time SVM hardware architecture developed through this body of work. With this funding an Altera Stratix V DSP development board was acquired. Figure 4.19 shows the Altera Stratix V DSP development board that succeeded the Terasic DE1 and DE0-Nano development boards and was used for all further research and development completed as part of this body of work.


***Figure 4.19:*** *Altera Stratix V DSP development Board.*

As discussed in Subsection 4.2.1, the primary research goal of this work was to achieve

real-time performance; all other metrics and parameters were initially unknown. These parameters were be investigated and explored in design variations. Also as discussed Subsection 4.2.1, the architecture was designed with the exploitation of massively parallel, systolic, and pipelined processing-element potential of FPGA-based DSP implementations.

The Stratix V FPGA family was at the time of the Altera Stratix V DSP development board acquisition the-top-of-the line and state-of-the-art Altera FPGA offering; the potential for utilising the technology's maximum specifications served the goal of this research project - to explore and implement a real-time SVM system. Thus the platform also served to provided a benchmark of measurable electrical and DSP-related parameters across several design variations. For example the requirement for embedded DSP and mathematical functional-block hardware far exceeds the need for raw implementable logic available on the chosen target device - a requirement the Altera Stratix V GS 5SGSMD5 device found on the Altera Stratix V DSP development board met.

Finally for future extensions of this work - whether commercial, industrial, or research based application - the Altera Stratix V device family serves as a good benchmark and guaranteed platform, in terms of device life-span and commercial availability, to improve upon system designs and thus improve derived and measured parameters and metrics gained through the scientific application of this body of work.

### 4.2.4 Ancillary Software Tools

In addition to Altera Quartus II / Quartus Prime other ancillary software was employed and utilised throughout the course of this research and development process. VHDL debugging and simulation was conducted using Mentor Graphic's ModelSim-Altera Edition simulation tool [89]. Algorithm and functional signal processing prototyping was conducted using MathWork's MATLAB [90] and the open-source equivalent GNU Octave [91]. Chaotic and non-linear time-series analysis was conducted using TISEAN: Nonlinear Time Series Analysis tools [92]. All `c` software and software models were compiled with the `gcc` [93] toolchain and run on Arch Linux [94] systems. All source code was managed using the Distributed Version Control System (DVCS) `git` [95] and all repositories were hosted privately on Atlassian Bitbucket [96] DVCS hosting service.

This thesis was written using LaTeX [97]. All original plots found in this thesis were generated with GNUPlot [98] and all original figures were created using the vector drawing tool `ipe` [99].

### 4.2.5 System Design Considerations

Rajkumar's proposed SVM system model is shown from a data-flow perspective in Fig. 4.20. The function blocks shown in grey, *SVM Training* and *SVM Model Evaluation*, were the primary concern of this work's research, design, and development efforts. This

included the implementation of a generalised FPGA hardware test-rig platform that can be used to validate and test developed SVM hardware modules in both real-time and in batch-processing mode. It was also required that the hardware platform be implemented in a modular and extensible manner to allow for future research-driven extensions and development with minimal VHDL refactoring requirements by undergraduate students and research engineers with minimal correspondence with the original author.

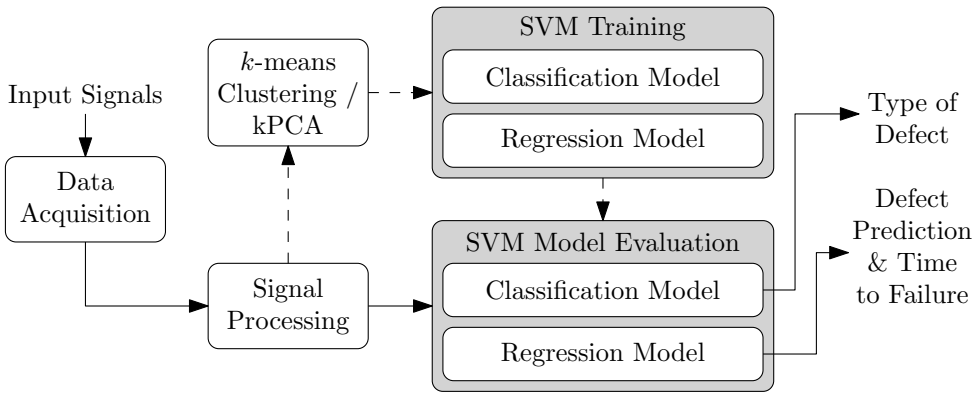



***Figure 4.20:*** *Simplified data-flow model of Rajkumar's original work.*

The hardware test-rig platform design and implementation conforms to the same data-flow model as Rajkumar's proposed system shown in Fig. 4.20.

### 4.2.6 SVM Test-Rig Design and Implementation

Figure 4.21 provides a system-level block-diagram of the FPGA hardware SVM test-rig system and supplementary PC Terminal implementation and testing infrastructure systems that were developed and employed as part of this body of work. The test-rig hardware system has allowed for the thorough testing and rigorous application of each developed SVM DSP pipeline. The SVM DSP pipeline subsystem architectures are elaborated upon in Section 4.3.

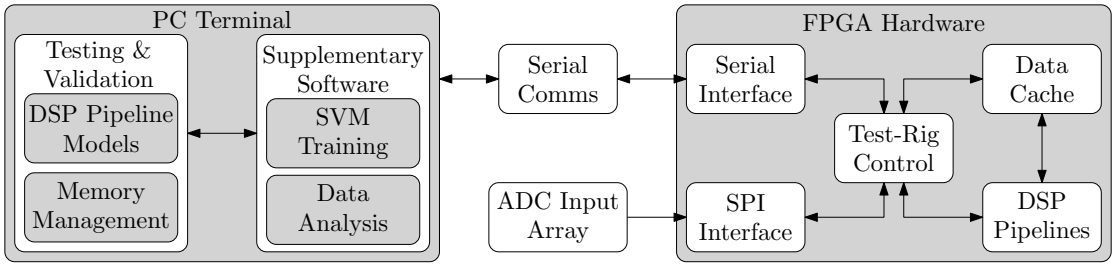



***Figure 4.21:*** *Top-level block-diagram of the FPGA hardware test-rig system and supplementary software subsystems.*

The system shown in Fig. 4.21 has been implemented utilising pipelined computation and data flow structures wherever possible and appropriate to maximise the test-rig system's overall data throughput capability and performance on the Altera Stratix V FPGA hardware platform.

Figure. 4.22 provides a low-level hardware architectural overview of the developed SVM test-rig system. The SVM test-rig system design incorporates a shared data bus with individually mapped data bus lines to each subsystem module The shared data bus is managed by a simple bus arbitration subsystem. Each test-rig subsystem is controlled by the test-rig's command and control subsystem. The test-rig system utilises three separate and distinct `uart` cores - one `uart` is dedicated to system control and the other two `uart` cores, `db0` and `db1`, are available for low-level hardware debugging and reporting apparatus. An input data cache `in_cache` is utilised to store SVM training and function evaluation parameters and batch processing data. An output data cache `out_cache` is utilised to store processed output data. The DSP pipeline module contains one of the many SVM DSP pipelines developed as part of this work; the SVM DSP pipeline architectures are discussed in Section 4.3. Finally the test-bench system design includes a 32-channel SPI array core to receive sampled real-world process data via an externally connected 32-channel ADC hardware array.

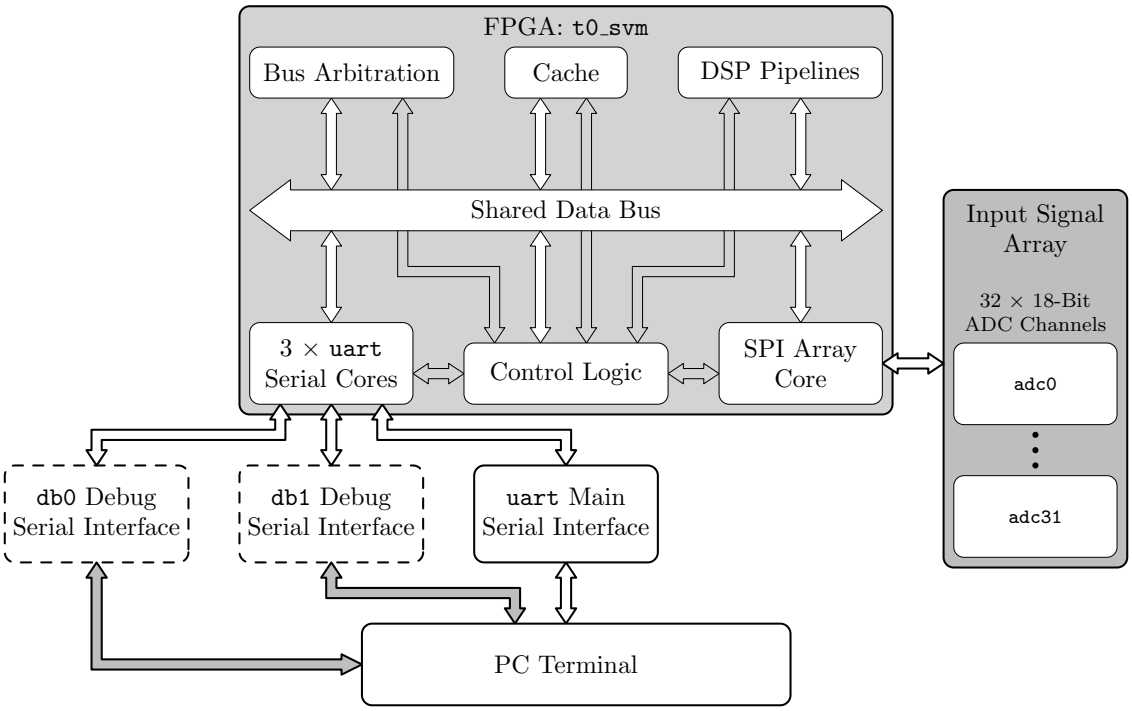



***Figure 4.22:*** *Test-rig hardware system architectural overview.*

The Altera Quartus Prime development suite was used to perform all circuit compilation, optimisation, synthesis, and Altera Stratix V GS 5SGSMD5 FPGA device fitting and subsequent programming as per the full set of developed VHDL code listings generated through this work's design and development efforts. The hierarchical VHDL module dependency tree shown in Fig. 4.23 provides an overview of the system's VHDL codebase and its modular structure.

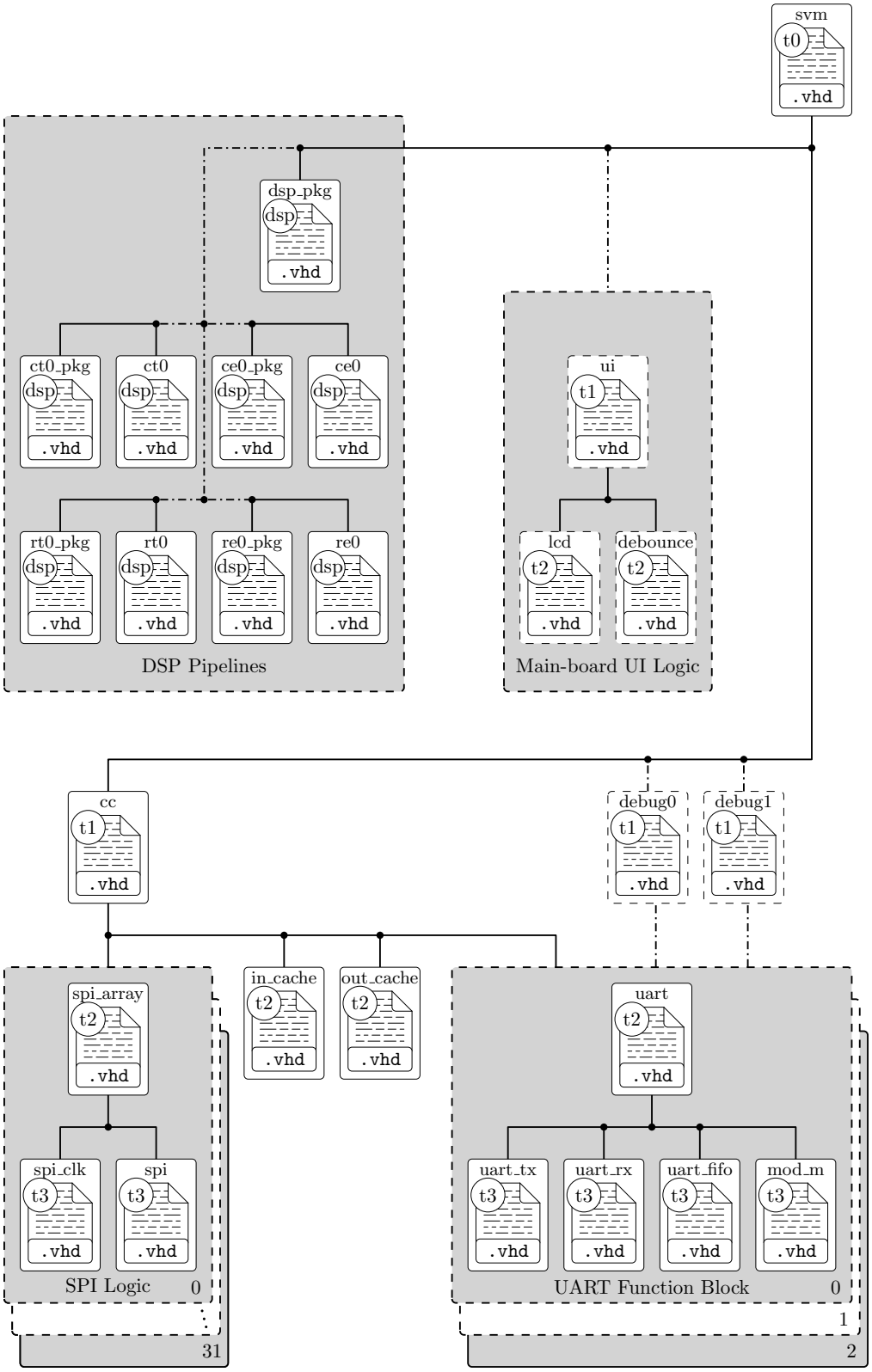



***Figure 4.23:*** *Test-rig system VHDL module dependency tree.*

The SVM test-rig system is controlled externally via a PC terminal over a simple serial port connection. By issuing appropriate hexadecimal commands followed by appropriately sized and formatted data packets one can control the test-rig system's state of execution

and process data accordingly. Figure 4.24 illustrates the test-rig system's command and control finite state machine including each state's hexadecimal command, state name and control function, and the control relationships between each.

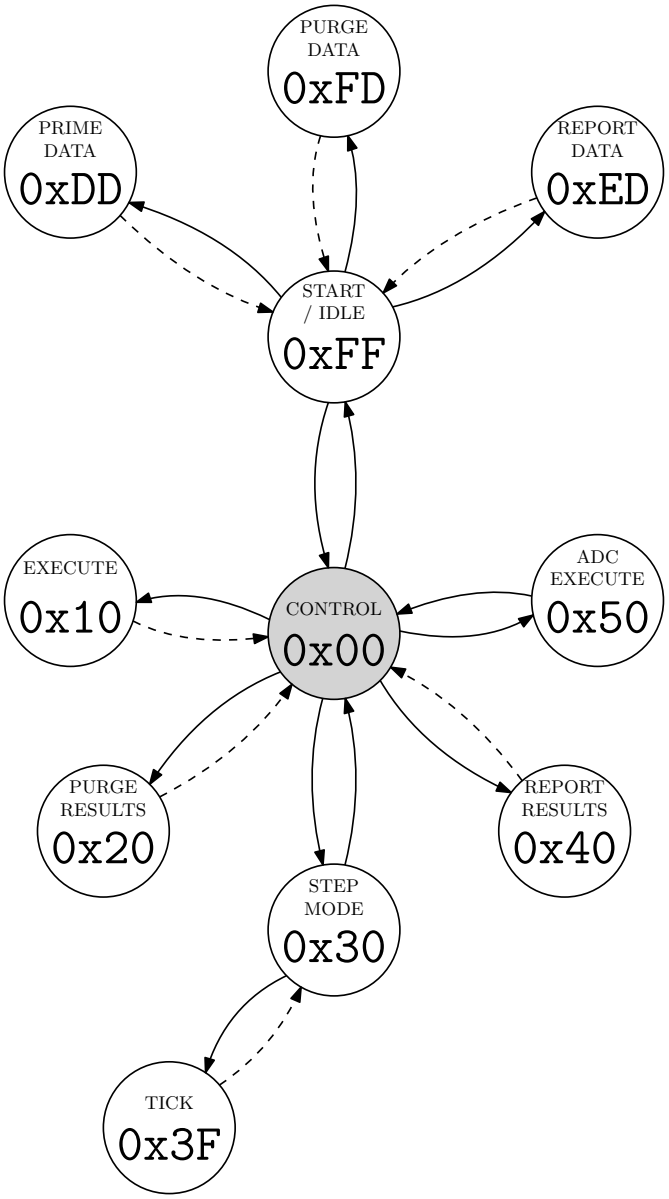



***Figure 4.24:*** *Test-rig system command and control finite state machine.*

Figure 4.25 illustrates the generalised bit-mapping of the SVM test-rig system's input data cache `in_cache`; input data cache size and bit-mapped address layout is dependant on the DSP pipeline implementation incorporated into the test-rig system.

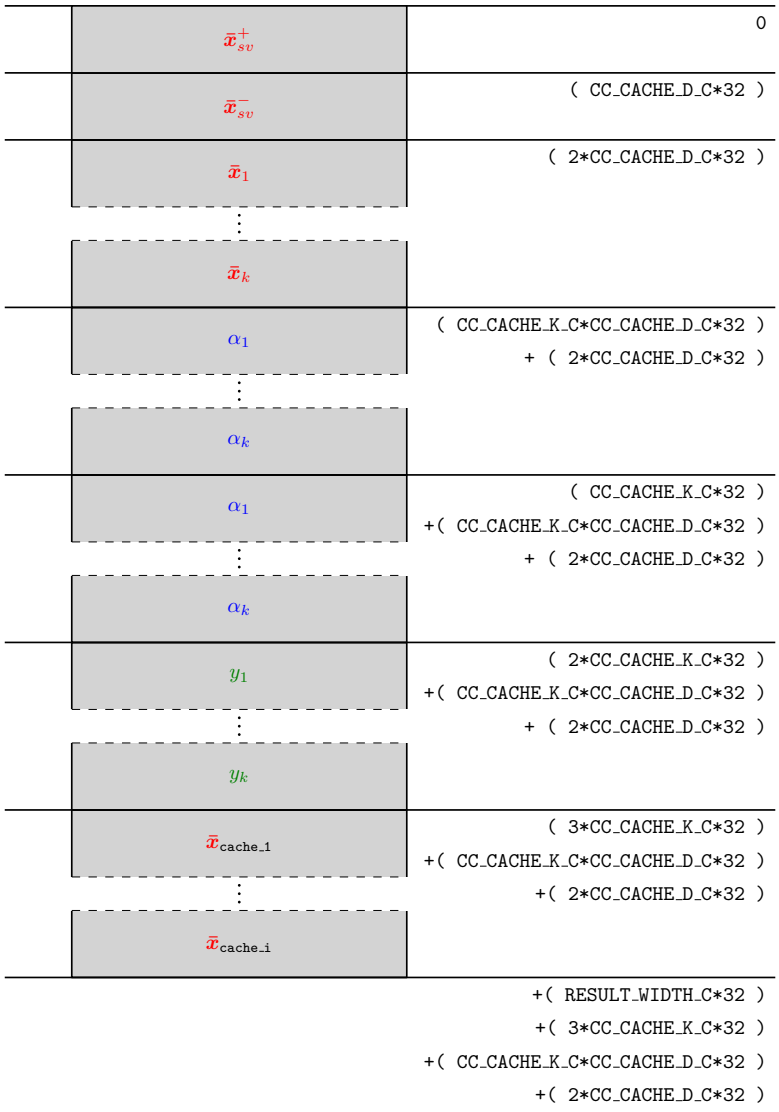



***Figure 4.25:*** *Test-rig system serial input data cache map.*

Figure 4.26 illustrates the generalised bit-mapping of the SVM test-rig system's result output data cache `out_cache`.

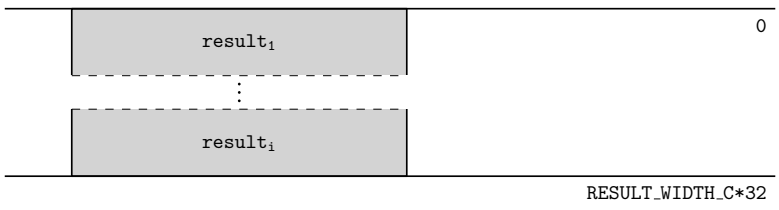



***Figure 4.26:*** *Test-rig system result cache map.*

## 4.3 SVM DSP Pipelines

This section presents the SVM DSP pipelines and kernel pipelines that were designed and implemented as part of this body of work; included in this section are technical details of the theory of operation, adaptation of the underlying computational structures for physical FPGA hardware realisation, and, an overview of each DSP pipeline architecture. DSP pipelines implementing *Classification Training*, *Classification Evaluation*, *Regression Training*, and *Regression Evaluation* have also been modelled in the C programming language and compiled to run on Intel x86 and ARMv7 microprocessor architecture Arch Linux platforms, and, implemented in VHDL, synthesised, and fitted to an Altera Stratix V GS FPGA device. The polynomial kernel is used in all of these SVM pipelines. Other kernel pipeline designs are presented in Appendix B.

Table 4.1 provides a list of the of `ct0`. Classification Training Pipelines implemented in both VHDL for the Altera Startix V FPGA and modelled C. Table 4.2 provides a list of the of `ce0`. Classification Evaluation Pipelines implemented in both VHDL for the Altera Startix V FPGA and modelled C. Table 4.3 provides a list of the of `rt0`. Regression Training Pipelines implemented in both VHDL for the Altera Startix V FPGA and modelled C. Table 4.4 provides a list of the of `re0`. Regression Evaluation Pipelines implemented in both VHDL for the Altera Startix V FPGA and modelled C.

***Table 4.1:*** *List of* `ct0`*. Classification Training Pipelines implemented in* VHDL *for the Altera Startix V* FPGA *and modelled in c.*

| Pipeline Name | Dimensions | Support Vectors |
|---|---|---|
| dsp_d2_k4_ct0 | 2 | 4 |
| dsp_d2_k8_ct0 | | 8 |
| dsp_d2_k16_ct0 | | 16 |
| dsp_d2_k32_ct0 | | 32 |
| dsp_d4_k8_ct0 | 4 | 8 |
| dsp_d4_k16_ct0 | | 16 |
| dsp_d4_k32_ct0 | | 32 |
| dsp_d8_k16_ct0 | 8 | 16 |
| dsp_d8_k32_ct0 | | 32 |

***Table 4.2:*** *List of* `ce0`*. Classification Evaluation Pipelines implemented in* VHDL *for the Altera Startix V* FPGA *and modelled in c.*

| Pipeline Name | Dimensions | Support Vectors |
|---|---|---|
| dsp_d2_k4_ce0 | 2 | 4 |
| dsp_d2_k8_ce0 | | 8 |
| dsp_d2_k16_ce0 | | 16 |
| dsp_d2_k32_ce0 | | 32 |
| dsp_d4_k8_ce0 | 4 | 8 |
| dsp_d4_k16_ce0 | | 16 |
| dsp_d4_k32_ce0 | | 32 |
| dsp_d8_k16_ce0 | 8 | 16 |
| dsp_d8_k32_ce0 | | 32 |

***Table 4.3:*** *List of* `rt0`*. Regression Training Pipelines implemented in* VHDL *for the Altera Startix V* FPGA *and modelled in c.*

| Pipeline Name | Dimensions | Support Vectors |
|---|---|---|
| dsp_d2_k4_rt0 | 2 | 4 |
| dsp_d2_k8_rt0 | | 8 |
| dsp_d2_k16_rt0 | | 16 |
| dsp_d2_k32_rt0 | | 32 |
| dsp_d4_k8_rt0 | 4 | 8 |
| dsp_d4_k16_rt0 | | 16 |
| dsp_d4_k32_rt0 | | 32 |
| dsp_d8_k16_rt0 | 8 | 16 |
| dsp_d8_k32_rt0 | | 32 |

***Table 4.4:*** *List of* `re0`*. Regressions Evaluation Pipelines implemented in* VHDL *for the Altera Startix V* FPGA *and modelled in c.*

| Pipeline Name | Dimensions | Support Vectors |
|---|---|---|
| dsp_d2_k4_re0 | 2 | 4 |
| dsp_d2_k8_re0 | | 8 |
| dsp_d2_k16_re0 | | 16 |
| dsp_d2_k32_re0 | | 32 |
| dsp_d4_k8_re0 | 4 | 8 |
| dsp_d4_k16_re0 | | 16 |
| dsp_d4_k32_re0 | | 32 |
| dsp_d8_k16_re0 | 8 | 16 |
| dsp_d8_k32_re0 | | 32 |

As an example of the nested mathematical structures each DSP pipeline is composed, the linear kernel and polynomial kernel Register Transfer Logic (RTL) architectural structure is presented here. Each kernel can be decomposed into an RTL representation as shown in Fig. 4.27 for the linear kernel and Fig. 4.28 for the polynomial kernel respectively. The input vectors $\bar{\boldsymbol{a}}$ and $\bar{\boldsymbol{b}}$ have an even number of elements $n$. If $\bar{\boldsymbol{a}}$ and $\bar{\boldsymbol{b}}$ do not have an even number of elements $n$, 0 can be used for the inputs.

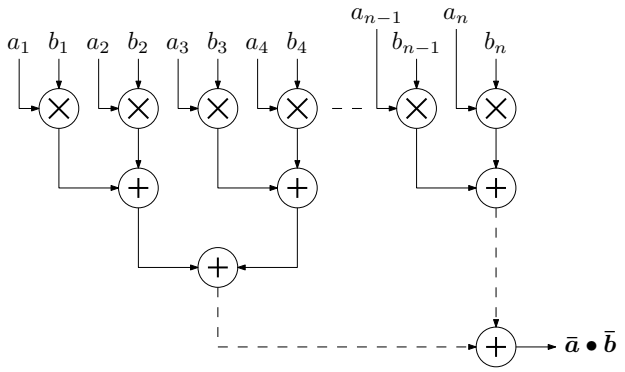



***Figure 4.27:*** *General RTL architectural structure of the linear kernel evaluation operation.*

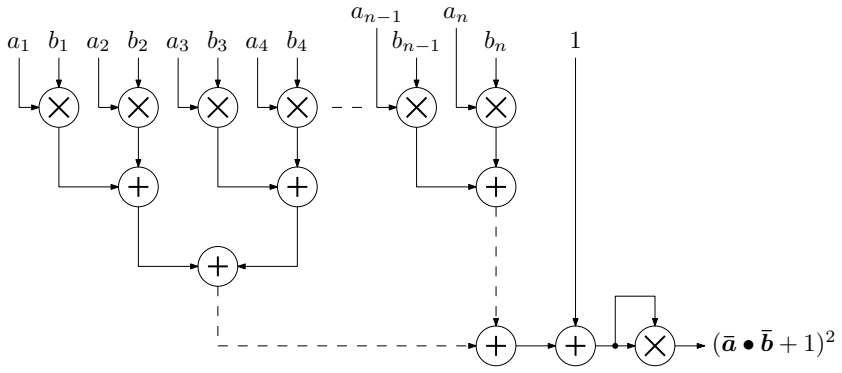
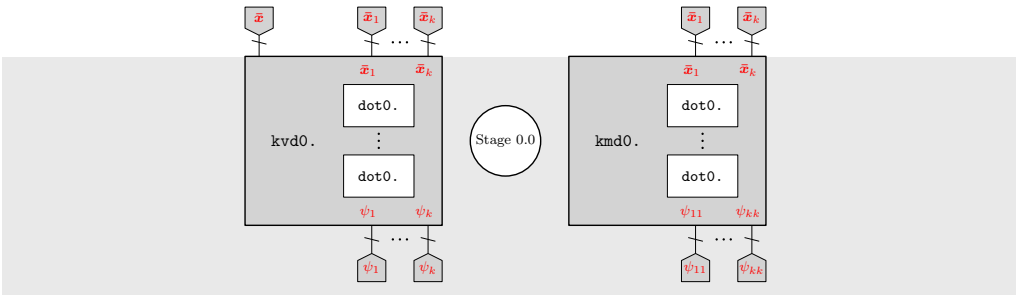



***Figure 4.28:*** *General RTL architectural structure of the polynomial kernel evaluation operation.*

Each successive layer of RTL operations shown in Fig. 4.27 and Fig. 4.28 are implemented as pipelined stages. Figure 4.29 illustrates the Linear kernel as a pipeline of staged-instructions. Table 4.5 lists the Linear kernel pipeline instruction set and corresponding implemented operations. Similarly, Figure 4.30 illustrates the Polynomial kernel as a pipeline of staged-instructions. Table 4.6 lists the Polynomial kernel pipeline instruction set and corresponding implemented operations. Stages shown in grey are drawn as one single stage to simplify pipeline stage synchronisation. The grey stages are, however, implemented as a series of stages where the number of stages in the series is a function of input data dimensionality. The grey stage convention is used for all pipelines shown throughout this section.




$$\psi(\bar{\boldsymbol{x}}_i, \bar{\boldsymbol{x}}_j) = \bar{\boldsymbol{x}}_i \bullet \bar{\boldsymbol{x}}_j$$

***Figure 4.29:*** *Linear kernel pipeline.*

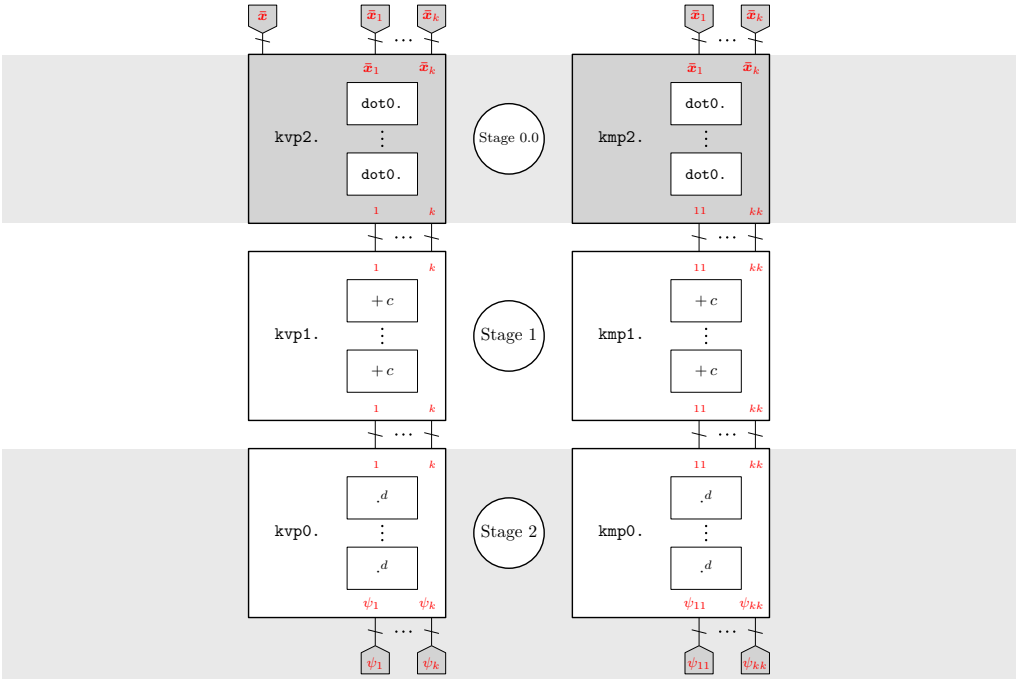
***Table 4.5:*** *Linear kernel pipeline instruction overview.*

| Instruction | Mathematical Operation |
|---|---|
| | $\psi(\bar{\boldsymbol{x}}_i, \bar{\boldsymbol{x}}) \Rightarrow \bar{\boldsymbol{\psi}}_{i\times 1} = \bar{\boldsymbol{\psi}}$ |
| kvd0. | $\Rightarrow \begin{bmatrix} \bar{\boldsymbol{x}}_1 \cdot \bar{\boldsymbol{x}} \\ \vdots \\ \bar{\boldsymbol{x}}_k \cdot \bar{\boldsymbol{x}} \end{bmatrix}$ |
| | $\psi(\bar{\boldsymbol{x}}_i, \bar{\boldsymbol{x}}_j) \Rightarrow \boldsymbol{\psi}_{i\times j} = \boldsymbol{\psi}$ |
| kmd0. | $\Rightarrow \begin{bmatrix} \bar{\boldsymbol{x}}_1 \cdot \bar{\boldsymbol{x}}_1 & \cdots & \bar{\boldsymbol{x}}_1 \cdot \bar{\boldsymbol{x}}_k \\ \vdots & \ddots & \vdots \\ \bar{\boldsymbol{x}}_k \cdot \bar{\boldsymbol{x}}_1 & \cdots & \bar{\boldsymbol{x}}_k \cdot \bar{\boldsymbol{x}}_k \end{bmatrix}$ |

Listing 4.1 provides the VHDL required to instantiate the two linear kernel pipeline variations shown in Fig. 4.29 and Table 4.5. All DSP pipeline hardware stage implementations presented in the this thesis utilise the same VHDL code structure as that shown in Listing 4.1 thus further DSP pipeline VHDL listings will not be included in this thesis. VHDL Entitiy listings for each implemented pipeline, however, are presented in Appendix C.

***Listing 4.1:*** *VHDL code listing: Linear kernel pipeline stage.*

```
1  -- kvd0
2  kvd0_stage00 : FOR i IN 0 TO K_C-1 GENERATE
3  kvd0_stage00_dot0 : PROCESS( clk, rst, pipe_en_s(ST00_C) ) IS
4
5  VARIABLE kvd0_v : SIGNED( BITS_C-1 DOWNTO 0 );
6
7  BEGIN
8    IF rst = '1' THEN -- asynchronous reset
9      kvd0_s(i) <= ( OTHERS => '0' );
10   ELSIF RISING_EDGE( clk ) THEN
11     IF pipe_en_s(ST00_C) = '1' THEN
12       kvd0_v := RESIZE( (SIGNED( xv_s(D_C*i) ) * SIGNED( x_s(0) )), BITS_C ) +
             RESIZE( (SIGNED( xv_s((D_C*i)+1) ) * SIGNED( x_s(1) )), BITS_C );
13     ELSE
14       kvd0_v := TO_SIGNED(0, BITS_C);
15      END IF;
16     kvd0_s(i) <= STD_LOGIC_VECTOR(kvd0_v);
17   END IF;
18
19 END PROCESS kvd0_stage00_dot0;
20 END GENERATE kvd0_stage00;
21
22 -- kmd0.
23 kmd0i_stage00 : FOR i IN 0 TO K_C-1 GENERATE
24 kmd0j_stage00 : FOR j IN 0 TO K_C-1 GENERATE
25 kmd0_stage00_dot0 : PROCESS( clk, rst, pipe_en_s(ST00_C) ) IS
26
27 VARIABLE kmd0_v : SIGNED( BITS_C-1 DOWNTO 0 );
28
29 BEGIN
30   IF rst = '1' THEN -- asynchronous reset
31     kmd0_s((K_C*i)+j) <= ( OTHERS => '0' );
32   ELSIF RISING_EDGE( clk ) THEN
33     IF pipe_en_s(ST00_C) = '1' THEN
34       kmd0_v := RESIZE( (SIGNED( xv_s(D_C*i) ) * SIGNED( xv_s(D_C*j) )), BITS_C ) +
            RESIZE( (SIGNED( xv_s((D_C*i)+1) ) * SIGNED( xv_s((D_C*j)+1) )), BITS_C
           );
35     ELSE
36       kmd0_v := TO_SIGNED(0, BITS_C);
37     END IF;
38     kmd0_s((K_C*i)+j) <= STD_LOGIC_VECTOR(kmd0_v);
39   END IF;
40
41 END PROCESS kmd0_stage00_dot0;
42 END GENERATE kmd0j_stage00;
43 END GENERATE kmd0i_stage00;
```




$$\psi(\bar{\boldsymbol{x}}_i, \bar{\boldsymbol{x}}_j) = (\bar{\boldsymbol{x}}_i \bullet \bar{\boldsymbol{x}}_j + c)^d$$

***Figure 4.30:*** *Polynomial kernel pipeline.*

***Table 4.6:*** *Polynomial kernel pipeline instruction overview.*

| Instruction | Mathematical Operation |
|---|---|
| | $\psi(\bar{\boldsymbol{x}}_i, \bar{\boldsymbol{x}}) \Rightarrow \bar{\boldsymbol{\psi}}_{i\times 1} = \bar{\boldsymbol{\psi}}$ |
| kvp2. | $\Rightarrow \begin{bmatrix} \bar{\boldsymbol{x}}_1 \cdot \bar{\boldsymbol{x}} \\ \vdots \\ \bar{\boldsymbol{x}}_k \cdot \bar{\boldsymbol{x}} \end{bmatrix}$ |
| kvp1. | $\Rightarrow \begin{bmatrix} [\bar{\boldsymbol{x}}_1 \cdot \bar{\boldsymbol{x}}] + c \\ \vdots \\ [\bar{\boldsymbol{x}}_k \cdot \bar{\boldsymbol{x}}] + c \end{bmatrix}$ |
| kvp0. | $\Rightarrow \begin{bmatrix} ([\bar{\boldsymbol{x}}_1 \cdot \bar{\boldsymbol{x}}] + c)^d \\ \vdots \\ ([\bar{\boldsymbol{x}}_k \cdot \bar{\boldsymbol{x}}] + c)^d \end{bmatrix} = \begin{bmatrix} \psi_1 \\ \vdots \\ \psi_k \end{bmatrix} = \bar{\boldsymbol{\psi}}$ |
| | $\psi(\bar{\boldsymbol{x}}_i, \bar{\boldsymbol{x}}_j) \Rightarrow \boldsymbol{\psi}_{i\times j} = \boldsymbol{\psi}$ |
| kmp2. | $\Rightarrow \begin{bmatrix} \bar{\boldsymbol{x}}_1 \cdot \bar{\boldsymbol{x}}_1 & \cdots & \bar{\boldsymbol{x}}_1 \cdot \bar{\boldsymbol{x}}_k \\ \vdots & \ddots & \vdots \\ \bar{\boldsymbol{x}}_k \cdot \bar{\boldsymbol{x}}_1 & \cdots & \bar{\boldsymbol{x}}_k \cdot \bar{\boldsymbol{x}}_k \end{bmatrix}$ |
| kmp1. | $\Rightarrow \begin{bmatrix} [\bar{\boldsymbol{x}}_1 \cdot \bar{\boldsymbol{x}}_1] + c & \cdots & [\bar{\boldsymbol{x}}_1 \cdot \bar{\boldsymbol{x}}_k] + c \\ \vdots & \ddots & \vdots \\ [\bar{\boldsymbol{x}}_k \cdot \bar{\boldsymbol{x}}_1] + c & \cdots & [\bar{\boldsymbol{x}}_k \cdot \bar{\boldsymbol{x}}_k] + c \end{bmatrix}$ |
| kmp0. | $\Rightarrow \begin{bmatrix} ([\bar{\boldsymbol{x}}_1 \cdot \bar{\boldsymbol{x}}_1] + c)^d & \cdots & ([\bar{\boldsymbol{x}}_1 \cdot \bar{\boldsymbol{x}}_k] + c)^d \\ \vdots & \ddots & \vdots \\ ([\bar{\boldsymbol{x}}_k \cdot \bar{\boldsymbol{x}}_1] + c)^d & \cdots & ([\bar{\boldsymbol{x}}_k \cdot \bar{\boldsymbol{x}}_k] + c)^d \end{bmatrix} = \begin{bmatrix} \psi_{11} & \cdots & \psi_{1k} \\ \vdots & \ddots & \vdots \\ \psi_{k1} & \cdots & \psi_{kk} \end{bmatrix} = \boldsymbol{\psi}$ |

Figure 4.31 illustrates the `ct0.` classification training pipeline with the polynomial kernel. Table 4.7 lists the `ct0.` classification training pipeline instruction set and corresponding implemented operations.

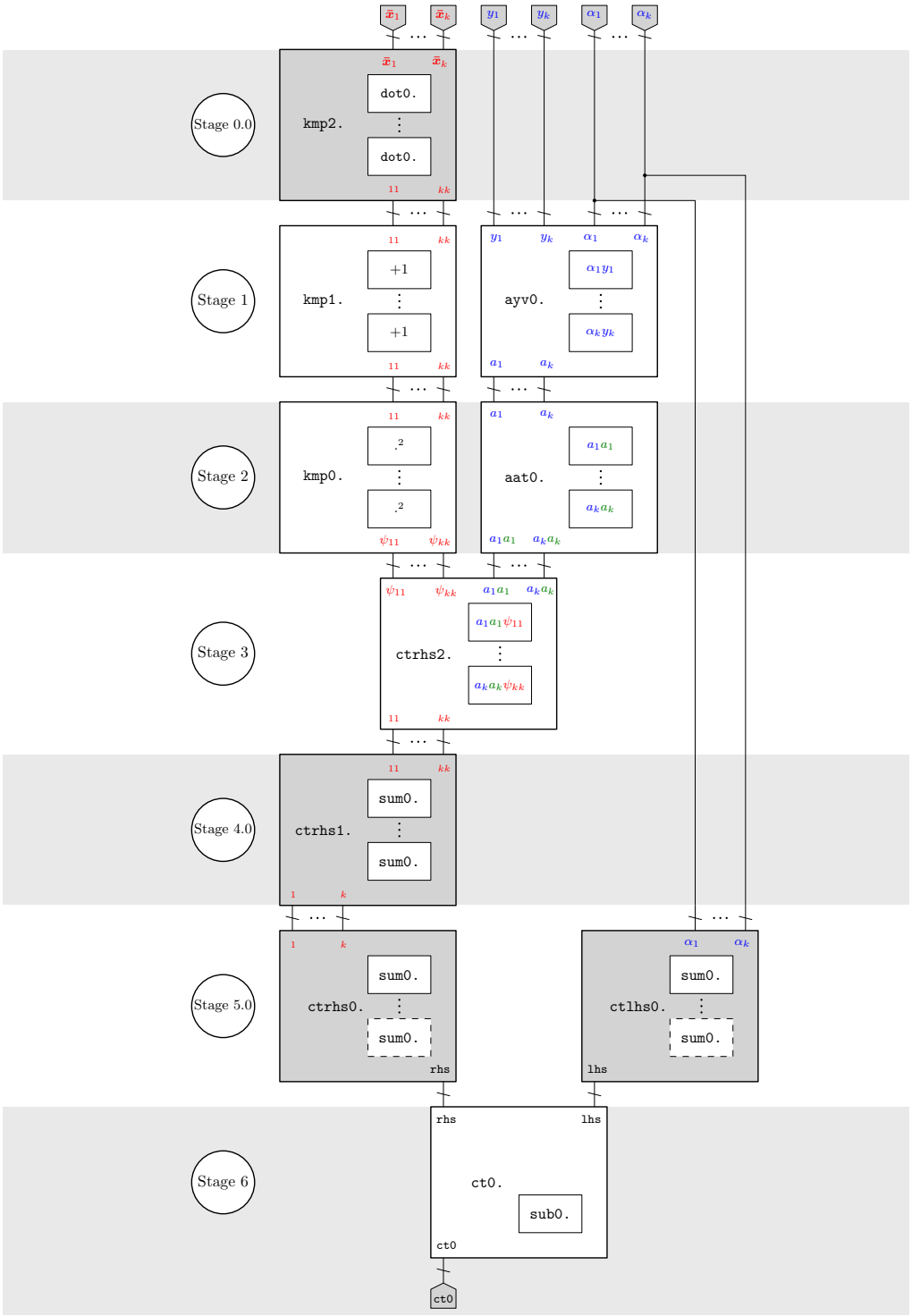



***Figure 4.31:*** `ct0.` *Classification Training Pipeline.*

***Table 4.7:*** `ct0.` *Classification Training Pipeline instruction overview.*

| Instruction | Mathematical Operation |
|---|---|
| | $\psi(\bar{\boldsymbol{x}}_i, \bar{\boldsymbol{x}}_j) \Rightarrow \boldsymbol{\psi}_{i\times j} = \boldsymbol{\psi}$ |
| `kmp2.` | $\Rightarrow \begin{bmatrix} \bar{\boldsymbol{x}}_1 \cdot \bar{\boldsymbol{x}}_1 & \cdots & \bar{\boldsymbol{x}}_1 \cdot \bar{\boldsymbol{x}}_k \\ \vdots & \ddots & \vdots \\ \bar{\boldsymbol{x}}_k \cdot \bar{\boldsymbol{x}}_1 & \cdots & \bar{\boldsymbol{x}}_k \cdot \bar{\boldsymbol{x}}_k \end{bmatrix}$ |
| `kmp1.` | $\Rightarrow \begin{bmatrix} [\bar{\boldsymbol{x}}_1 \cdot \bar{\boldsymbol{x}}_1] + 1 & \cdots & [\bar{\boldsymbol{x}}_1 \cdot \bar{\boldsymbol{x}}_k] + 1 \\ \vdots & \ddots & \vdots \\ [\bar{\boldsymbol{x}}_k \cdot \bar{\boldsymbol{x}}_1] + 1 & \cdots & [\bar{\boldsymbol{x}}_k \cdot \bar{\boldsymbol{x}}_k] + 1 \end{bmatrix}$ |
| `kmp0.` | $\Rightarrow \begin{bmatrix} ([\bar{\boldsymbol{x}}_1 \cdot \bar{\boldsymbol{x}}_1] + 1)^2 & \cdots & ([\bar{\boldsymbol{x}}_1 \cdot \bar{\boldsymbol{x}}_k] + 1)^2 \\ \vdots & \ddots & \vdots \\ ([\bar{\boldsymbol{x}}_k \cdot \bar{\boldsymbol{x}}_1] + 1)^2 & \cdots & ([\bar{\boldsymbol{x}}_k \cdot \bar{\boldsymbol{x}}_k] + 1)^2 \end{bmatrix} = \begin{bmatrix} \boldsymbol{\psi}_{11} & \cdots & \boldsymbol{\psi}_{1k} \\ \vdots & \ddots & \vdots \\ \boldsymbol{\psi}_{k1} & \cdots & \boldsymbol{\psi}_{kk} \end{bmatrix} = \boldsymbol{\psi}$ |
| `ayv0.` | $\Rightarrow \bar{\boldsymbol{a}} = [\bar{\boldsymbol{a}}]_{\,i\times 1} = \bar{\boldsymbol{a}} = [\bar{\boldsymbol{a}}]_{\,i\times 1} = \begin{bmatrix} \alpha_1 y_1 \\ \vdots \\ \alpha_k y_k \end{bmatrix} = \begin{bmatrix} a_1 \\ \vdots \\ a_k \end{bmatrix}$ |
| `aat0.` | $\Rightarrow \bar{\boldsymbol{a}}\bar{\boldsymbol{a}}^T = \left[\bar{\boldsymbol{a}}\bar{\boldsymbol{a}}^T\right]_{i\times i} = \begin{bmatrix} a_1 a_1 & \cdots & a_1 a_k \\ \vdots & \ddots & \vdots \\ a_k a_1 & \cdots & a_k a_k \end{bmatrix}$ |
| | $\frac{1}{2}\sum_{i=1}^{k}\sum_{j=1}^{k}\alpha_i\alpha_j y_i y_j \psi(\bar{\boldsymbol{x}}_i, \bar{\boldsymbol{x}}_j) = \frac{1}{2}\bar{\boldsymbol{a}}^T \boldsymbol{\psi}\, \bar{\boldsymbol{a}}$ |
| `ctrhs2.` | $\Rightarrow \begin{bmatrix} a_1 a_1 \boldsymbol{\psi}_{11} & \cdots & a_1 a_k \boldsymbol{\psi}_{1k} \\ \vdots & \ddots & \vdots \\ a_k a_1 \boldsymbol{\psi}_{k1} & \cdots & a_k a_k \boldsymbol{\psi}_{kk} \end{bmatrix}$ |
| `ctrhs1.` | $\Rightarrow \begin{bmatrix} [a_1 a_1 \boldsymbol{\psi}_{11}] + \cdots + [a_1 a_k \boldsymbol{\psi}_{1k}] \\ \vdots \\ [a_k a_1 \boldsymbol{\psi}_{k1}] + \cdots + [a_k a_k \boldsymbol{\psi}_{kk}] \end{bmatrix}$ |
| `ctrhs0.` | $\Rightarrow \quad [a_1 a_1 \boldsymbol{\psi}_{11}] + \cdots + [a_1 a_k \boldsymbol{\psi}_{1k}] + \cdots + [a_k a_1 \boldsymbol{\psi}_{k1}] + \cdots + [a_k a_k \boldsymbol{\psi}_{kk}]$<br>The 1/2 operation is a right-shift of 1 bit at the output of `ctrhs0.` stage. |
| `ctlhs0.` | $\Rightarrow \sum_{i=1}^{k} \alpha_i = \alpha_1 + \cdots + \alpha_k$ |
| `ct0.` | $\Rightarrow \left(\sum_{i=1}^{k} \alpha_i\right) - \left(\frac{1}{2}\sum_{i=1}^{k}\sum_{j=1}^{k}\alpha_i\alpha_j y_i y_j \psi(\bar{\boldsymbol{x}}_i, \bar{\boldsymbol{x}}_j)\right) =$ `ctlhs0. − ctrhs0.` |

Figure 4.32 illustrates the `ce0.` classification evaluation pipeline with the polynomial kernel. Table 4.8 lists the `ce0.` classification evaluation pipeline instruction set and corresponding implemented operations.

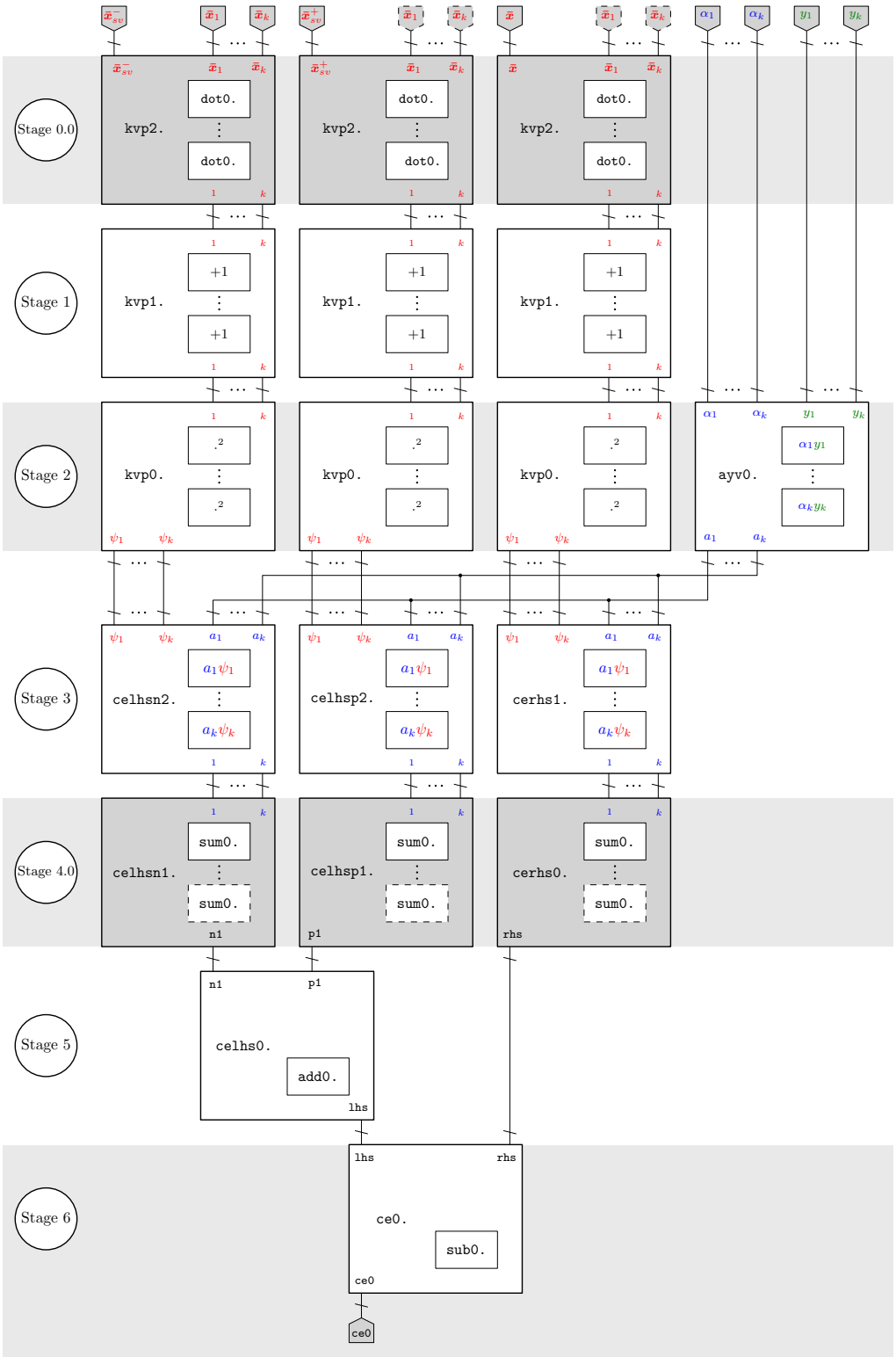



***Figure 4.32:*** `ce0`. *Classification Evaluation Pipeline.*

***Table 4.8:*** `ce0.` *Classification Evaluation DSP Pipeline instruction overview.*

| Instruction | Mathematical Operation |
|---|---|
| | $\psi(\bar{\boldsymbol{x}}_i, \bar{\boldsymbol{x}}) \Rightarrow \bar{\boldsymbol{\psi}}_{i\times 1} = \bar{\boldsymbol{\psi}}$ |
| `kvp2.` | $\Rightarrow \begin{bmatrix} \bar{\boldsymbol{x}}_1 \cdot \bar{\boldsymbol{x}} \\ \vdots \\ \bar{\boldsymbol{x}}_k \cdot \bar{\boldsymbol{x}} \end{bmatrix}$ |
| `kvp1.` | $\Rightarrow \begin{bmatrix} [\bar{\boldsymbol{x}}_1 \cdot \bar{\boldsymbol{x}}] + 1 \\ \vdots \\ [\bar{\boldsymbol{x}}_k \cdot \bar{\boldsymbol{x}}] + 1 \end{bmatrix}$ |
| `kvp0.` | $\Rightarrow \begin{bmatrix} ([\bar{\boldsymbol{x}}_1 \cdot \bar{\boldsymbol{x}}] + 1)^2 \\ \vdots \\ ([\bar{\boldsymbol{x}}_k \cdot \bar{\boldsymbol{x}}] + 1)^2 \end{bmatrix} = \begin{bmatrix} \psi_1 \\ \vdots \\ \psi_k \end{bmatrix} = \bar{\boldsymbol{\psi}}$ |
| `ayv0.` | $\Rightarrow \bar{\boldsymbol{a}} = [\bar{\boldsymbol{a}}]_{i\times 1} = \begin{bmatrix} \alpha_1 y_1 \\ \vdots \\ \alpha_k y_k \end{bmatrix} = \begin{bmatrix} a_1 \\ \vdots \\ a_k \end{bmatrix}$ |
| `celhsp2.` `celhsn2.` `cerhs1.` | $\Rightarrow \begin{bmatrix} a_1\psi_1 \\ \vdots \\ a_k\psi_k \end{bmatrix}$ |
| `celhsp1.` `celhsn1.` `cerhs0.` | $\Rightarrow \sum_{i=1}^{k} \alpha_i y_i \psi(\bar{\boldsymbol{x}}_i, \bar{\boldsymbol{x}}) = \bar{\boldsymbol{a}}^T \bar{\boldsymbol{\psi}} = [a_1\psi_1] + \cdots + [a_k\psi_k]$ |
| | $\frac{1}{2}\left(\sum_{i=1}^{k} \alpha_i y_i \psi(\bar{\boldsymbol{x}}_i, \bar{\boldsymbol{x}}_{sv^+}) + \sum_{i=1}^{k} \alpha_i y_i \psi(\bar{\boldsymbol{x}}_i, \bar{\boldsymbol{x}}_{sv^-})\right)$ |
| `celhs0.` | $\Rightarrow$ `celhsp1.` + `celhsn1.`<br>The 1/2 operation is a right-shift of 1 bit at the output of `celhs0.` stage. |
| `ce0.` | $\Rightarrow \sum_{i=1}^{k} \alpha_i y_i \psi_i - \frac{1}{2}\left(\sum_{i=1}^{k} \alpha_i y_i \psi_{i^+} + \sum_{i=1}^{k} \alpha_i y_i \psi_{i^-}\right) =$ `cerhs0.` $-$ `celhs0.` |

Figure 4.33 illustrates the `rt0.` regression training pipeline with the polynomial kernel. Table 4.9 lists the `rt0.` regression training pipeline instruction set and corresponding implemented operations.

Figure 4.34 illustrates the `re0.` regression evaluation pipeline with the polynomial kernel. Table 4.10 lists the `re0.` regression evaluation pipeline instruction set and corresponding implemented operations.

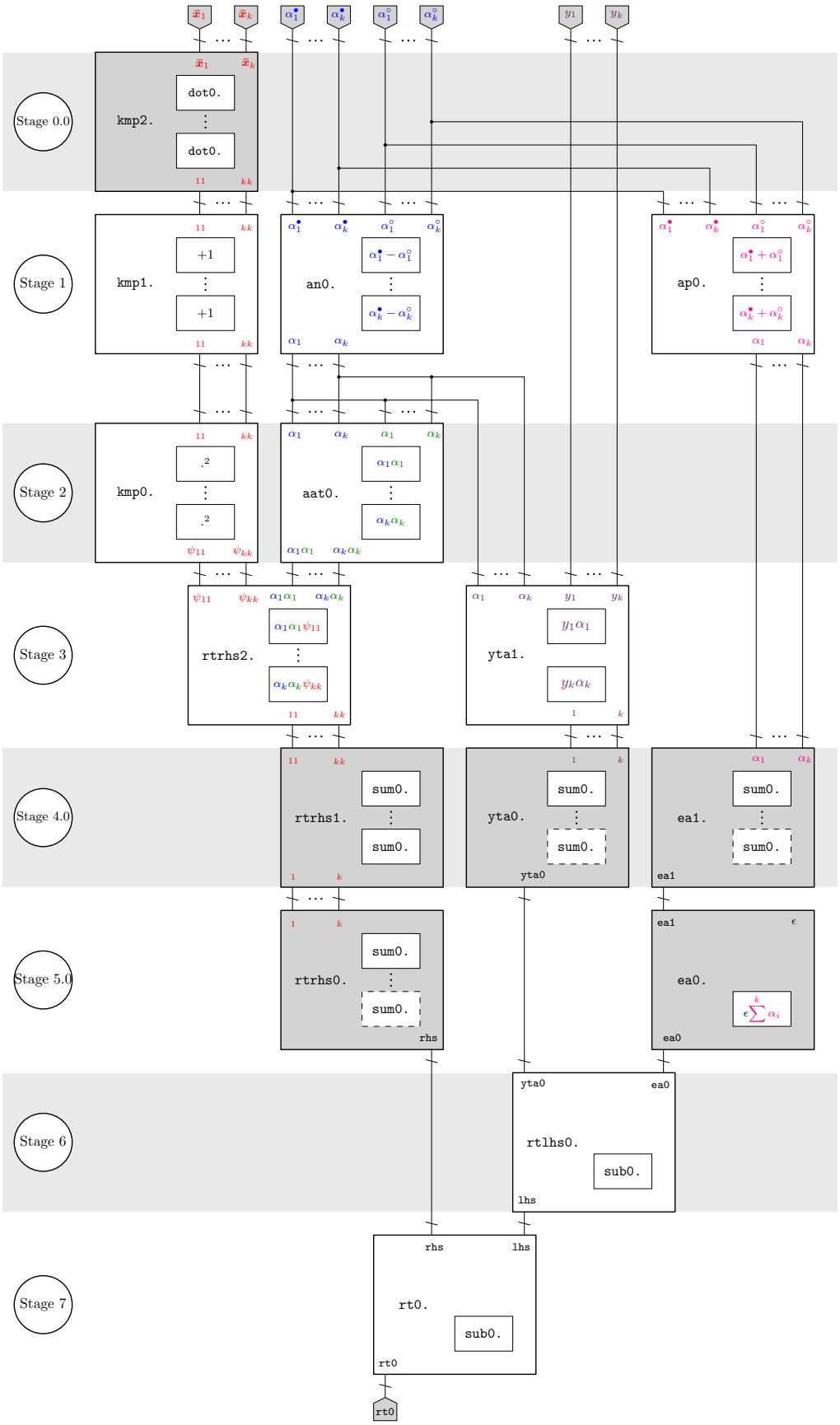



***Figure 4.33:*** *`rt0`. Regression Training Pipeline.*

***Table 4.9:*** *`rt0`. Regression Training DSP Pipeline instruction overview.*

| Instruction | Mathematical Operation |
|---|---|
| | $\psi(\bar{\boldsymbol{x}}_i, \bar{\boldsymbol{x}}_j) \Rightarrow \boldsymbol{\psi}_{i\times j} = \boldsymbol{\psi}$ |
| `kmp2.` | $\Rightarrow \begin{bmatrix} \bar{\boldsymbol{x}}_1 \cdot \bar{\boldsymbol{x}}_1 & \cdots & \bar{\boldsymbol{x}}_1 \cdot \bar{\boldsymbol{x}}_k \\ \vdots & \ddots & \vdots \\ \bar{\boldsymbol{x}}_k \cdot \bar{\boldsymbol{x}}_1 & \cdots & \bar{\boldsymbol{x}}_k \cdot \bar{\boldsymbol{x}}_k \end{bmatrix}$ |
| `kmp1.` | $\Rightarrow \begin{bmatrix} [\bar{\boldsymbol{x}}_1 \cdot \bar{\boldsymbol{x}}_1] + 1 & \cdots & [\bar{\boldsymbol{x}}_1 \cdot \bar{\boldsymbol{x}}_k] + 1 \\ \vdots & \ddots & \vdots \\ [\bar{\boldsymbol{x}}_k \cdot \bar{\boldsymbol{x}}_1] + 1 & \cdots & [\bar{\boldsymbol{x}}_k \cdot \bar{\boldsymbol{x}}_k] + 1 \end{bmatrix}$ |
| `kmp0.` | $\Rightarrow \begin{bmatrix} ([\bar{\boldsymbol{x}}_1 \cdot \bar{\boldsymbol{x}}_1] + 1)^2 & \cdots & ([\bar{\boldsymbol{x}}_1 \cdot \bar{\boldsymbol{x}}_k] + 1)^2 \\ \vdots & \ddots & \vdots \\ ([\bar{\boldsymbol{x}}_k \cdot \bar{\boldsymbol{x}}_1] + 1)^2 & \cdots & ([\bar{\boldsymbol{x}}_k \cdot \bar{\boldsymbol{x}}_k] + 1)^2 \end{bmatrix} = \begin{bmatrix} \boldsymbol{\psi}_{11} & \cdots & \boldsymbol{\psi}_{1k} \\ \vdots & \ddots & \vdots \\ \boldsymbol{\psi}_{k1} & \cdots & \boldsymbol{\psi}_{kk} \end{bmatrix} = \boldsymbol{\psi}$ |
| `ap0.` | $\Rightarrow \bar{\boldsymbol{\alpha}}^{\bullet} + \bar{\boldsymbol{\alpha}}^{\circ} = \bar{\boldsymbol{\alpha}}_{i\times 1} = \bar{\boldsymbol{\alpha}}$ |
| `an0.` | $\Rightarrow \bar{\boldsymbol{\alpha}}^{\bullet} - \bar{\boldsymbol{\alpha}}^{\circ} = \bar{\boldsymbol{\alpha}}_{i\times 1} = \bar{\boldsymbol{\alpha}} = \bar{\boldsymbol{\alpha}}_{i\times 1} = \bar{\boldsymbol{\alpha}} = \bar{\boldsymbol{\alpha}}_{i\times 1} = \bar{\boldsymbol{\alpha}}$ |
| `aat0.` | $\bar{\boldsymbol{\alpha}}\bar{\boldsymbol{\alpha}}^T = \left[\bar{\boldsymbol{\alpha}}\bar{\boldsymbol{\alpha}}^T\right]_{i\times i} = \begin{bmatrix} \alpha_1\alpha_1 & \cdots & \alpha_1\alpha_k \\ \vdots & \ddots & \vdots \\ \alpha_k\alpha_1 & \cdots & \alpha_k\alpha_k \end{bmatrix}$ |
| | $\frac{1}{2}\sum_{i=1}^{k}\sum_{j=1}^{k}\alpha_i\alpha_j\psi(\bar{\boldsymbol{x}}_i, \bar{\boldsymbol{x}}_j) = \frac{1}{2}\bar{\boldsymbol{\alpha}}^T\boldsymbol{\psi}\,\bar{\boldsymbol{\alpha}}$ |
| `rtrhs2.` | $\Rightarrow \begin{bmatrix} \alpha_1\alpha_1\boldsymbol{\psi}_{11} & \cdots & \alpha_1\alpha_k\boldsymbol{\psi}_{1k} \\ \vdots & \ddots & \vdots \\ \alpha_k\alpha_1\boldsymbol{\psi}_{k1} & \cdots & \alpha_k\alpha_k\boldsymbol{\psi}_{kk} \end{bmatrix}$ |
| `rtrhs1.` | $\Rightarrow \begin{bmatrix} [\alpha_1\alpha_1\boldsymbol{\psi}_{11}] + \cdots + [\alpha_1\alpha_k\boldsymbol{\psi}_{1k}] \\ \vdots \\ [\alpha_k\alpha_1\boldsymbol{\psi}_{k1}] + \cdots + [\alpha_k\alpha_k\boldsymbol{\psi}_{kk}] \end{bmatrix}$ |
| `rtrhs0.` | $\Rightarrow [\alpha_1\alpha_1\boldsymbol{\psi}_{11}] + \cdots + [\alpha_1\alpha_k\boldsymbol{\psi}_{1k}] + \cdots + [\alpha_k\alpha_1\boldsymbol{\psi}_{k1}] + \cdots + [\alpha_k\alpha_k\boldsymbol{\psi}_{kk}]$<br>The 1/2 operation is a right-shift of 1 bit at the output of `rtrhs0.` stage. |
| | $\sum_{j=1}^{k} y_i\alpha_i - \epsilon\sum_{i=1}^{k}\alpha_i = \bar{\boldsymbol{y}}^T\bar{\boldsymbol{\alpha}} - \epsilon\left[\alpha_1 + \cdots + \alpha_k\right]$ |
| `ea1.` | $\Rightarrow \alpha_1 + \cdots + \alpha_k$ |
| `ea0.` | $\Rightarrow \epsilon\left[\alpha_1 + \cdots + \alpha_k\right]$ |
| `yta1.` | $\Rightarrow \begin{bmatrix} y_1\alpha_1 \\ \vdots \\ y_k\alpha_k \end{bmatrix}$ |
| `yta0.` | $\Rightarrow [y_1\alpha_1] + \cdots + [y_k\alpha_k]$ |
| `rtlhs0.` | $\Rightarrow [y_1\alpha_1] + \cdots + [y_k\alpha_k] - \epsilon\left[\alpha_1 + \cdots + \alpha_k\right] =$ `yta0. - ea0.` |
| `rt0.` | $\Rightarrow \left(\sum_{i=1}^{k} y_i\alpha_i - \epsilon\sum_{i=1}^{k}\alpha_i\right) - \left(\frac{1}{2}\sum_{i=1}^{k}\sum_{j=1}^{k}\alpha_i\alpha_j\psi(\bar{\boldsymbol{x}}_i, \bar{\boldsymbol{x}}_j)\right) =$ `rtlhs0. - rtrhs0.` |

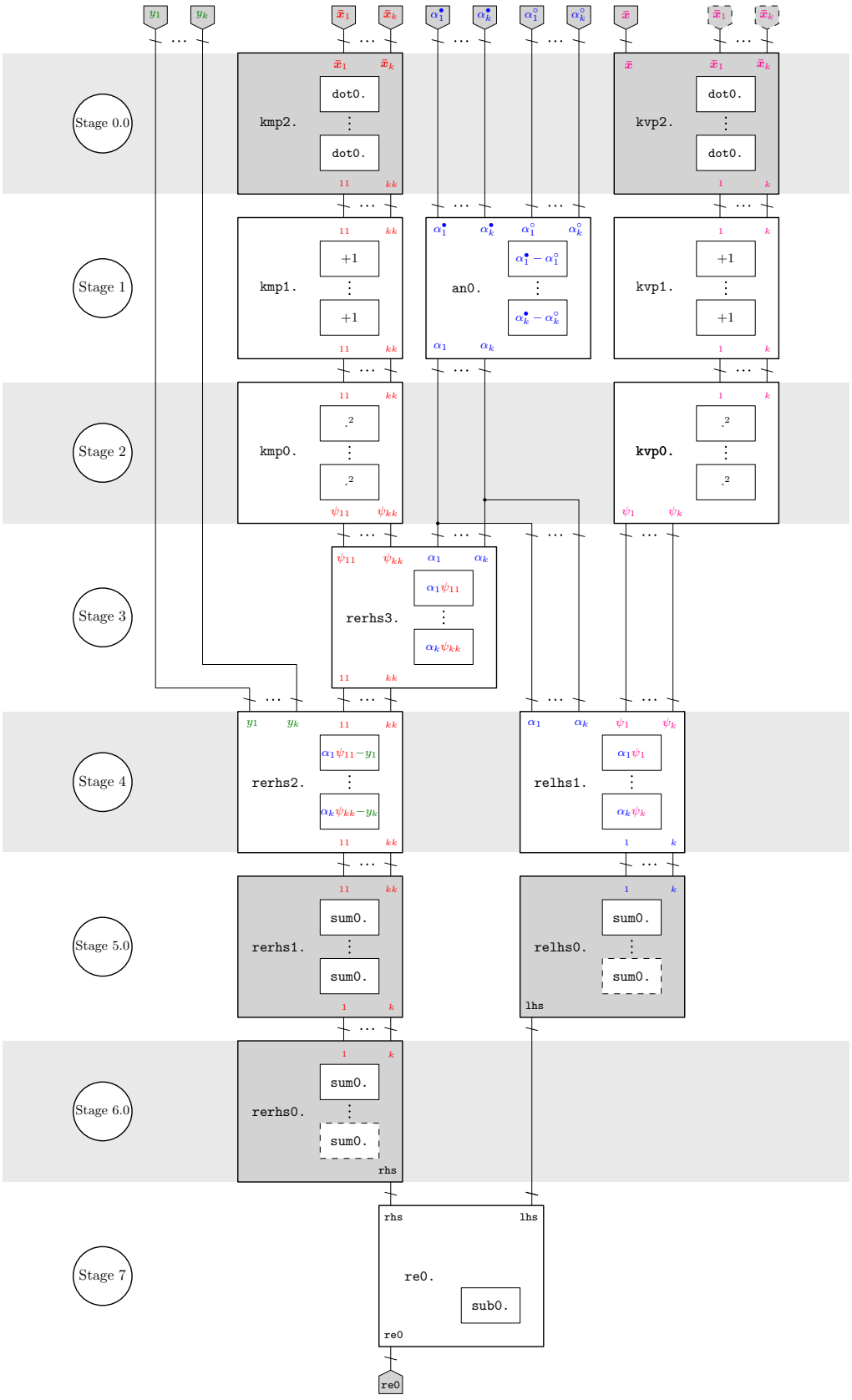



***Figure 4.34:*** `re0`. *Regression Evaluation Pipeline.*

***Table 4.10:*** `re0.` *Regression Evaluation DSP Pipeline instruction overview.*

| Instruction | Mathematical Operation |
|---|---|
| | $\psi(\bar{\boldsymbol{x}}_i, \bar{\boldsymbol{x}}) \Rightarrow \bar{\boldsymbol{\psi}}_{i\times 1} = \bar{\boldsymbol{\psi}}$ |
| `kvp2.` | $\Rightarrow \begin{bmatrix} \bar{\boldsymbol{x}}_1 \cdot \bar{\boldsymbol{x}} \\ \vdots \\ \bar{\boldsymbol{x}}_k \cdot \bar{\boldsymbol{x}} \end{bmatrix}$ |
| `kvp1.` | $\Rightarrow \begin{bmatrix} [\bar{\boldsymbol{x}}_1 \cdot \bar{\boldsymbol{x}}] + 1 \\ \vdots \\ [\bar{\boldsymbol{x}}_k \cdot \bar{\boldsymbol{x}}] + 1 \end{bmatrix}$ |
| `kvp0.` | $\Rightarrow \begin{bmatrix} ([\bar{\boldsymbol{x}}_1 \cdot \bar{\boldsymbol{x}}] + 1)^2 \\ \vdots \\ ([\bar{\boldsymbol{x}}_k \cdot \bar{\boldsymbol{x}}] + 1)^2 \end{bmatrix} = \begin{bmatrix} \psi_1 \\ \vdots \\ \psi_k \end{bmatrix} = \bar{\boldsymbol{\psi}}$ |
| | $\psi(\bar{\boldsymbol{x}}_i, \bar{\boldsymbol{x}}_j) \Rightarrow \boldsymbol{\psi}_{i\times j} = \boldsymbol{\psi}$ |
| `kmp2.` | $\Rightarrow \begin{bmatrix} \bar{\boldsymbol{x}}_1 \cdot \bar{\boldsymbol{x}}_1 & \cdots & \bar{\boldsymbol{x}}_1 \cdot \bar{\boldsymbol{x}}_k \\ \vdots & \ddots & \vdots \\ \bar{\boldsymbol{x}}_k \cdot \bar{\boldsymbol{x}}_1 & \cdots & \bar{\boldsymbol{x}}_k \cdot \bar{\boldsymbol{x}}_k \end{bmatrix}$ |
| `kmp1.` | $\Rightarrow \begin{bmatrix} [\bar{\boldsymbol{x}}_1 \cdot \bar{\boldsymbol{x}}_1] + 1 & \cdots & [\bar{\boldsymbol{x}}_1 \cdot \bar{\boldsymbol{x}}_k] + 1 \\ \vdots & \ddots & \vdots \\ [\bar{\boldsymbol{x}}_k \cdot \bar{\boldsymbol{x}}_1] + 1 & \cdots & [\bar{\boldsymbol{x}}_k \cdot \bar{\boldsymbol{x}}_k] + 1 \end{bmatrix}$ |
| `kmp0.` | $\Rightarrow \begin{bmatrix} ([\bar{\boldsymbol{x}}_1 \cdot \bar{\boldsymbol{x}}_1] + 1)^2 & \cdots & ([\bar{\boldsymbol{x}}_1 \cdot \bar{\boldsymbol{x}}_k] + 1)^2 \\ \vdots & \ddots & \vdots \\ ([\bar{\boldsymbol{x}}_k \cdot \bar{\boldsymbol{x}}_1] + 1)^2 & \cdots & ([\bar{\boldsymbol{x}}_k \cdot \bar{\boldsymbol{x}}_k] + 1)^2 \end{bmatrix} = \begin{bmatrix} \boldsymbol{\psi}_{11} & \cdots & \boldsymbol{\psi}_{1k} \\ \vdots & \ddots & \vdots \\ \boldsymbol{\psi}_{k1} & \cdots & \boldsymbol{\psi}_{kk} \end{bmatrix} = \boldsymbol{\psi}$ |
| `an0.` | $\Rightarrow \bar{\boldsymbol{\alpha}}^{\bullet} - \bar{\boldsymbol{\alpha}}^{\circ} = \bar{\boldsymbol{\alpha}}_{i\times 1} = \bar{\boldsymbol{\alpha}}$ |
| | $\sum_{i=1}^{k} \alpha_i \psi_i = \bar{\boldsymbol{\alpha}}^T \bar{\boldsymbol{\psi}}$ |
| `relhs1.` | $\Rightarrow \begin{bmatrix} \alpha_1 \psi_1 \\ \vdots \\ \alpha_k \psi_k \end{bmatrix}$ |
| `relhs0.` | $\Rightarrow [\alpha_1 \psi_1] + \cdots + [\alpha_k \psi_k]$ |
| | $\frac{1}{k} \sum_{i=1}^{k} \sum_{j=1}^{k} \alpha_i \psi_{ij} - y_j$ |
| `rerhs3.` | $\Rightarrow \begin{bmatrix} \alpha_1 \boldsymbol{\psi}_{11} & \cdots & \alpha_1 \boldsymbol{\psi}_{1k} \\ \vdots & \ddots & \vdots \\ \alpha_k \boldsymbol{\psi}_{k1} & \cdots & \alpha_k \boldsymbol{\psi}_{kk} \end{bmatrix}$ |
| `rerhs2.` | $\Rightarrow \begin{bmatrix} \alpha_1 \boldsymbol{\psi}_{11} - y_1 & \cdots & \alpha_1 \boldsymbol{\psi}_{1k} - y_k \\ \vdots & \ddots & \vdots \\ \alpha_k \boldsymbol{\psi}_{k1} - y_1 & \cdots & \alpha_k \boldsymbol{\psi}_{kk} - y_k \end{bmatrix}$ |
| `rerhs1.` | $\Rightarrow \begin{bmatrix} [\alpha_1 \boldsymbol{\psi}_{11} - y_1] + \cdots + [\alpha_1 \boldsymbol{\psi}_{1k} - y_k] \\ \vdots \\ [\alpha_k \boldsymbol{\psi}_{k1} - y_1] + \cdots + [\alpha_k \boldsymbol{\psi}_{kk} - y_k] \end{bmatrix}$ |
| `rerhs0.` | $\Rightarrow [\alpha_1 \boldsymbol{\psi}_{11} - y_1] + \cdots + [\alpha_1 \boldsymbol{\psi}_{1k} - y_k] + \cdots + [\alpha_k \boldsymbol{\psi}_{k1} - y_1] + \cdots + [\alpha_k \boldsymbol{\psi}_{kk} - y_k]$<br>The 1/k operation is a right-shift of k bits at the output of `rerhs0.` stage. |
| `re0.` | $\Rightarrow \left( \sum_{i=1}^{k} \alpha_i \psi_i \right) - \left( \frac{1}{k} \sum_{i=1}^{k} \sum_{j=1}^{k} \alpha_i \psi_{ij} - y_j \right) =$ `relhs0.` $-$ `rerhs0.` |

## 4.4 Scientific Methodologies

This section presents scientific methodologies designed as part of this work, and, the data-sets used in the application of these methodologies. The aims of the application of these methodologies is twofold. Primarily the developed SVM DSP pipelines have been applied to Rajkumar's oil and pipeline domain application. Secondarily chaotic data-sets have been used in various machine learning endeavours to investigate and explore the SVM machine learning paradigm's potential for use as a tool in non-linear and chaotic system's theory and time-series analysis.

Kernel PCA and k-means clustering tools of the MLPACK Machine Learning Library [100] were used to reduce dimensionality. LIBSVM [101] was used to generate classification and regression training sets and provide best-case SVM function evaluation experimental benchmarks, and, inform further SVM DSP pipeline data-set application exploration.

### 4.4.1 Data-sets and Machine Learning Experimental Overview

Table 4.11 provides an overview of all data-sets and the SVM machine learning experiments conducted as part of this work.

***Table 4.11:*** *SVM machine learning experiment overview.*

| | Machine Learning Experimental Overview | |
|---|---|---|
| Data-set | SVM Classification | SVM Regression |
| **LPD**<br>Legacy Oil & Gas Pipeline Data<br>Dimension $d$: 2, 4 and 8 | **C-LPD**<br>5 Class Classification<br>Dimension $d$: 2, 4 and 8 | **R-LPD**<br>Arbitrary-Valued<br>Staircase Function<br>Dimension $d$: 2, 4 and 8 |
| **LAD**<br>Lorenz Attractor Data<br>Dimension $d$: 1, 2, 3, 4 and 8 | **C-LAD**<br>5 Class Classification<br>Dimension $d$: 2, 4 and 8 | **R-LAD**<br>Parameter Identification<br>$R = 25, \cdots, 33$<br>Dimension $d$: 2, 4 and 8 |
| **MGAD**<br>Mackey-Glass Attractor Data<br>Dimension $d$: 1, 2, 4 and 8 | **C-MGAD**<br>5 Class Classification<br>Dimension $d$: 2, 4 and 8 | **R-MGAD**<br>Parameter Identification<br>$\tau = 17, \cdots, 25$<br>Dimension $d$: 2, 4 and 8 |
| **ANND**<br>ANN Chaotic Oscillator Data<br>Dimension $d$: 1, 2, 4 and 8 | **C-ANND**<br>5 Class Classification<br>Dimension $d$: 2, 4 and 8 | **R-ANND**<br>Parameter Identification<br>$D = 200, \cdots, 360$<br>Dimension $d$: 2, 4 and 8 |

Figure 4.35 provides an overview of the Legacy Oil and Gas Pipeline Data-Set (LPD) in 3-dimensional data-space rotated through 360 ° at 90 ° increments. Figure 4.36 provides an overview of the Lorenz Attractor Data-Set (LAD) state-space response for system parameters $P = 10$, $B = 8/3$, and (a) $R = 25$, with initial conditions $x(0) = 0.0$, $y(0) = -0.1$, and $z(0) = 9.0$, through to (e) $R = 33$, with each previous system's final state as serving as initial conditions for the next. Figure 4.37 provides an overview of the Mackey-Glass Attractor Data-Set (MGAD) 2-dimensional state-space response for system

parameters $a = 0.2$, $b = 0.1$, $c = 10$ and (a) $\tau = 17$, increased at increments of 2 through to (e) $\tau = 25$. The state-space evolutions shown in Fig. 4.36 and Fig. 4.37 is illustrated as a transition from green to blue. Figure 4.38 provides an overview of the Artificial Neural Network Chaotic Oscillator Data-Set (ANND) time-series with the number of neurons held constant at $N = 10$ and the delay-line length increased at increments of 40 from (a) $D = 200$, through to (e) $D = 360$.

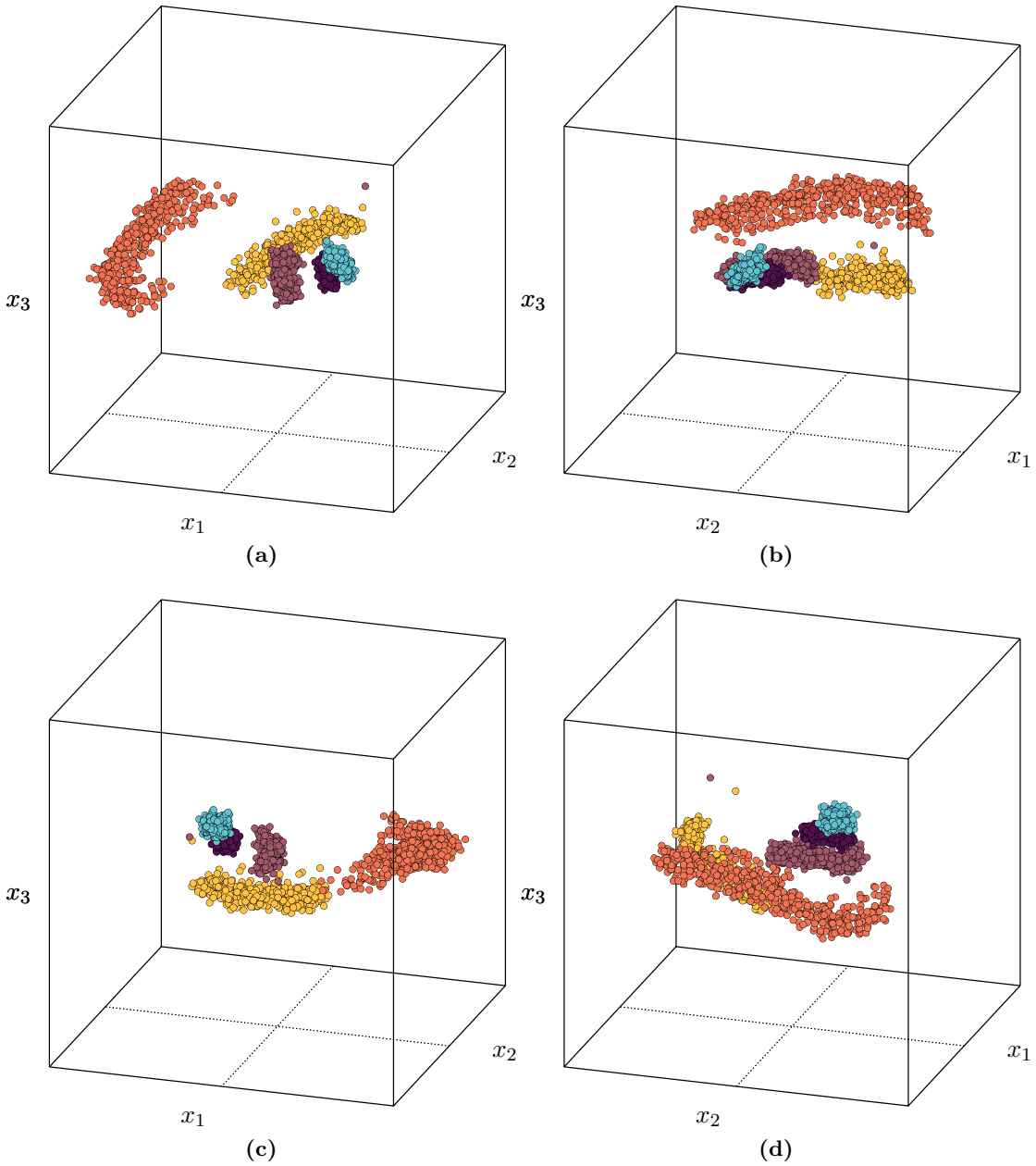



***Figure 4.35:*** *Legacy pipeline 3-dimensional data-space rotated through 360° at 90° increments.*

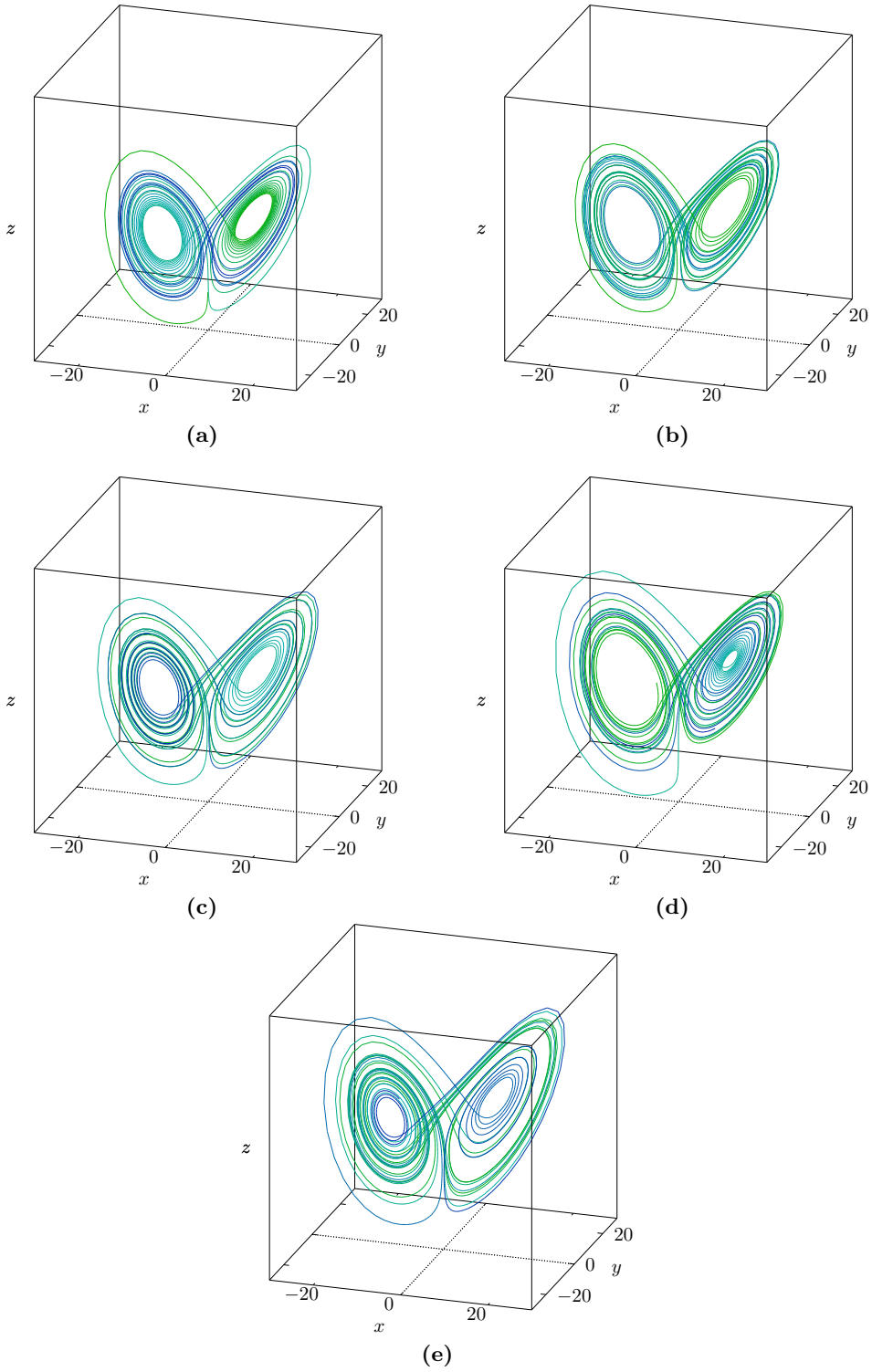



***Figure 4.36:*** *Lorenz Attractor state-space response for system parameters* $P = 10$, $B = 8/3$, *and (a)* $R = 25$, *with initial conditions* $x(0) = 0.0$, $y(0) = -0.1$, *and* $z(0) = 9.0$, *through to (e)* $R = 33$, *with each previous system's final state as initial conditions. Each state-space evolution is shown as a transition from green to blue.*

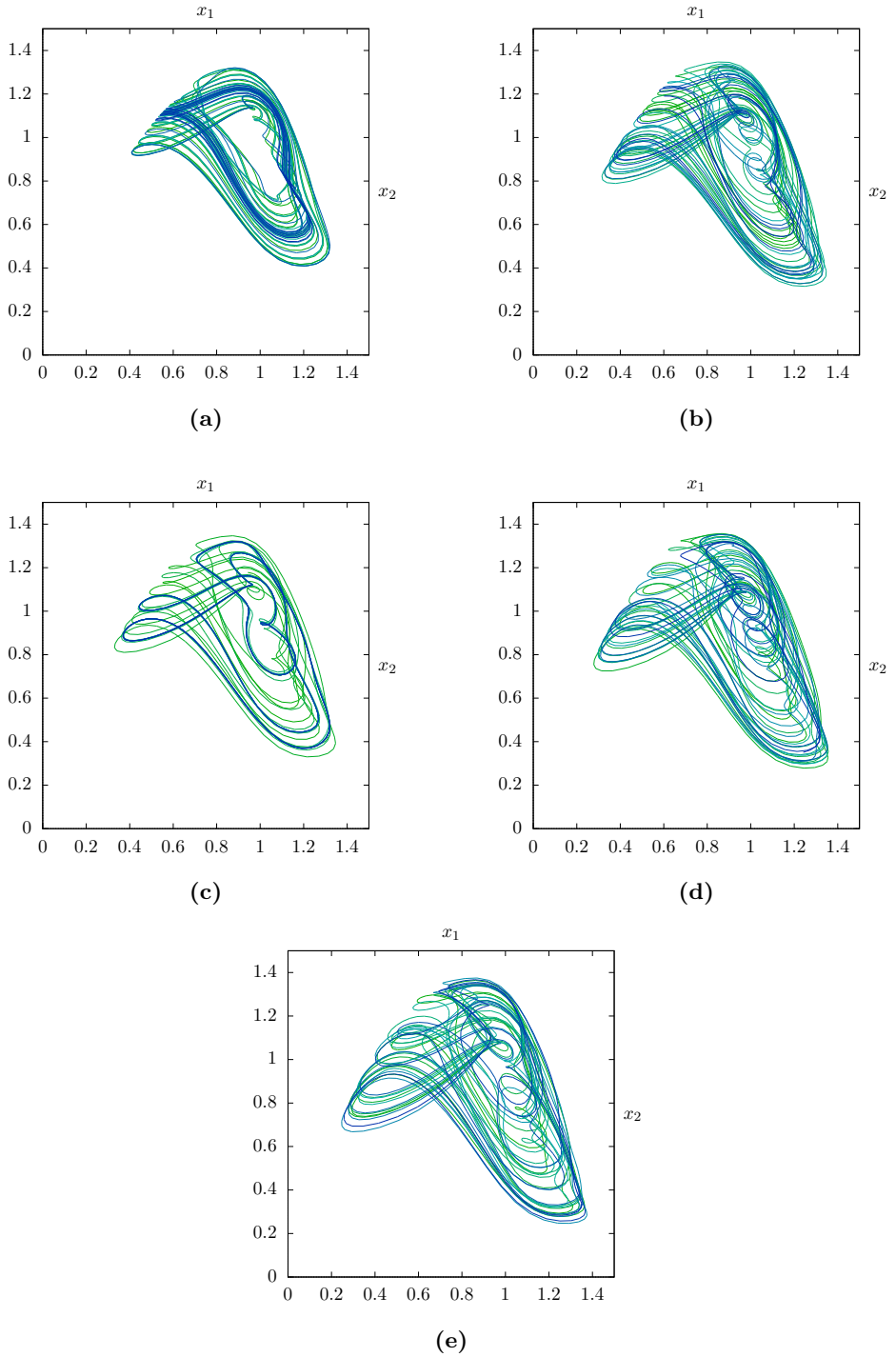



***Figure 4.37:*** *Mackey-Glass Attractor 2-dimensional state-space response for system parameters* $a = 0.2$, $b = 0.1$, $c = 10$ *and (a)* $\tau = 17$, *increased at increments of 2 through to (e)* $\tau = 25$. *Each state-space evolution is shown as a transition from green to blue.*

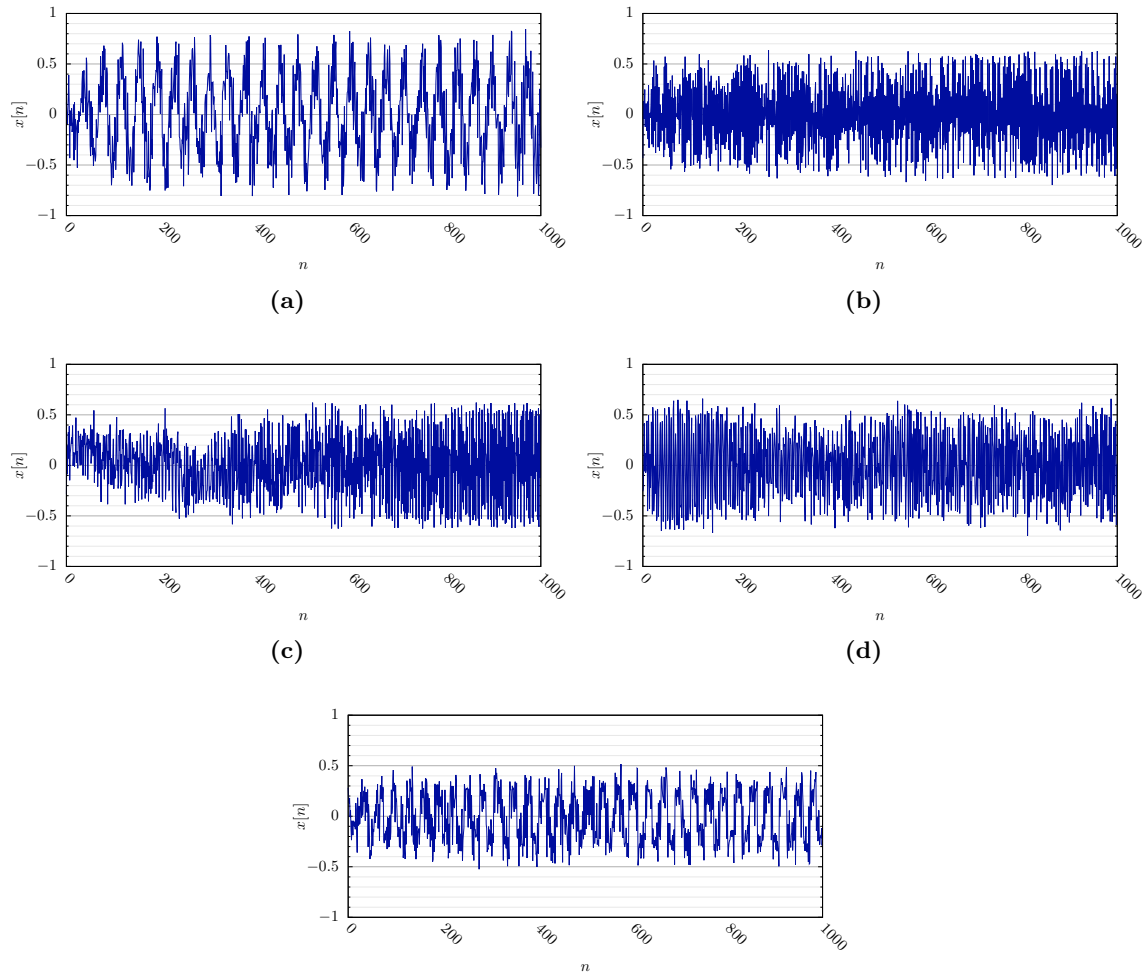



***Figure 4.38:*** *ANN Chaotic Oscillator time-series with the number of neurons held constant at* $N = 10$ *and the delay-line length increased at increments of 40 from (a)* $D = 200$*, through to (e)* $D = 360$*.*

### 4.4.2 Data-set Processing and Application of SVM Systems

Figure 4.39 provides a generalised overview of each data-set's data-flow through state-space embedding and kernel Principal Component Analysis (PCA) dimension reduction preprocessing strategies, training and data-set curation, SVM training strategies, and finally the application of SVM function evaluation implementations.

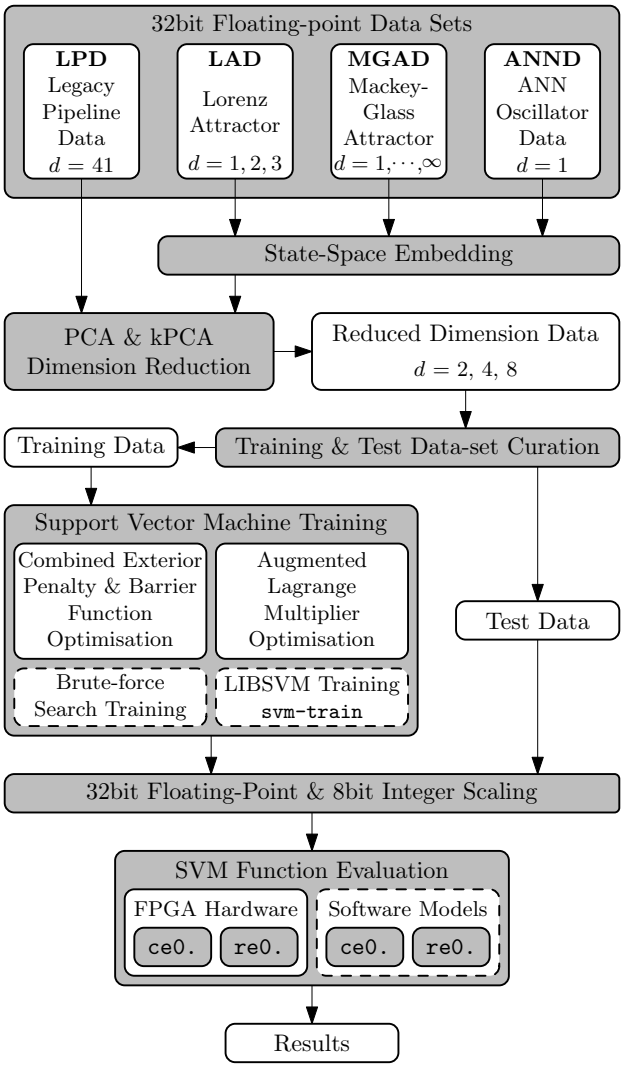



***Figure 4.39:*** *Experimental Data-set Processing Overview.*

#### 4.4.2.1 Legacy Pipeline Data Methodology

The LPD data-set was inherited from Rajkumar's original work [29]. The data-set in its original raw form is a 41 dimensions data-space. The dimensionality of the data-set was reduced using Kernel Principal Component Analysis (kPCA) with the polynomial kernel to a 2 dimension data-space, a 3 dimension data-space, a 4 dimension data-space, and a 8 dimension data-space. The 3 dimension data-space achieved through the kPCA process is shown in 4.35.

The reduced-dimension data-sets where then labelled appropriately and curated into training and test data-sets. The data curation process applied an equal-probability random sorting mechanic on each individual data-point into either a training data-set or a test data-set. For SVM classification, C-LPD, five distinct training-test data-set pairs were constructed; each training-test pair corresponded to a different -1 class data-cluster assignment and all remaining clusters were assigned to the +1 class. For SVM regression, R-LPD, each data-cluster was assigned an arbitrary enumerated label starting from 1

through to 5.

The training data-sets where then applied to each appropriate SVM classification and SVM regression training strategies to obtain $\alpha$ Lagrangian coefficients and support vector pair SVM model data. Finally the trained SVM models and test data-sets were then appropriately scaled and applied to the applicable SVM classification and SVM regression function evaluation system implementations.

#### 4.4.2.2 Chaotic Systems Data Methodology

The LAD data-set, the MGAD data-set, and the ANND data-set where generated through the execution of custom software implementations of each respective system. Observing the state-space evolutions shown in Fig. 4.36 and Fig. 4.37 and the time series shown in Fig. 4.38.

Variations (over a small interval) in each of the the chaotic system's underlying parameters does little to change the general structure of the chaotic attractors and the subsequent time-series and state-space evolution; this is clear by observing the state-space portraits shown in Fig. 4.36 and Fig. 4.37 and the time series shown in Fig. 4.38 and noting the very similar structures between each variation. This is after-all a key property of chaotic systems - a sensitive dependence to initial conditions. In order to introduce a more concrete relationship between each consecutive point along the state-space or time-series evolution - that is to embed more of the subtle variations in response between varied systems in each data-point, and, to increase the dissimilarities between each varied system or to increase the varied system's separation in feature-space, a 100 dimensional state-space embedding was applied to each of the data-sets and their variations.

The dimensionality of the 100 dimension embedded state-space chaotic system data-sets was reduced using kPCA with the polynomial kernel to a 2 dimension data-space, a 4 dimension data-space, and a 8 dimension data-space. The 3 dimension data-space achieved through the kPCA process is shown in 4.35.

The reduced-dimension data-sets where then labelled appropriately and curated into training and test data-sets. The data curation process applied an equal-probability random sorting mechanic on each individual data-point into either a training data-set or a test data-set. For SVM classification, C-LAD, C-MGAD, and C-ANND, five distinct training-test data-set pairs, for each chaotic system data-set, were constructed; each training-test pair corresponded to a different -1 class data-cluster assignment and all remaining clusters were assigned to the +1 class. For SVM regression, R-LAD, R-MGAD, and R-ANND, each data-cluster in each data-set was assigned a label reflecting the varied system parameter used to generate the original raw data. The Lorenz attractor system parameter $R$ was varied starting from $R = 25$ and increased at increments of 2 through to $R = 33$. The Mackey-Glass attractor system parameter $\tau$ was varied starting from $\tau = 17$ and in-

creased at increments of 2 through to $\tau = 25$. Finally the ANN chaotic oscillator system parameter $D$ was varied starting from $D = 200$ and were increased at increments of 40 through to $D = 360$.

The training data-sets where then applied to each appropriate SVM classification and SVM regression training strategies to obtain $\alpha$ Lagrangian coefficients and support vector pair SVM model data. Finally the trained SVM models and test data-sets were then appropriately scaled and applied to the applicable SVM classification and SVM regression function evaluation system implementations.

# Chapter 5

# Results

This chapter presents metrics gained through instrumentation and test measurement, and, experimentally-obtained results pertaining to the SVM systems designed and developed as part of the scope of this work and presented in Chapter 4. The chapter is organised into three distinct sections; Section 5.1 *DSP Results*, Section 5.2 *Electrical Results*, and Section 5.3 *Machine Learning Results*.

Section 5.1 *DSP Results* presents each pipeline architecture hardware implementation's FPGA resource utilisation, each pipeline architecture's hardware and software model latency and execution time, and each pipeline architecture's instruction-per-cycle metrics.

Section 5.2 *Electrical Results* presents each pipeline architecture hardware implementation's FPGA power consumption.

Section 5.3 *Machine Learning Results* presents results obtained through the application of this work's developed SVM systems with the four data-sets presented in Section 4.4. These results are arranged into two subsections - *Classification* and *Regression* - each dedicated to the machine learning experimental application of the same name as shown in Table 4.11 and outlined in Section 4.4. Each subsection is then further organised into sub-subsections by data-set: *Legacy Pipeline Data* (LPD), *Lorenz Attractor Data* (LAD), *Mackey-Glass Attractor Data* (MGAD), and *Artificial Neural Network Data* (ANND).

## 5.1 DSP Results

All DSP pipeline architecture FPGA hardware implementations where compiled and synthesised using *Altera Quartus Prime* [87] version 15.1 for an *Altera Stratix V GS 5SGSMD5* FPGA device [102], [103]. All DSP pipeline architecture software model implementations where compiled and executed on systems running the *Arch Linux* distribution [94] employing Linux kernel version 4.6.2 using `gcc` version 6.1.1 [93]. All software model executable metrics were collected using the Linux performance instrumentation and profiling tool `perf` [104].

Table 5.1 provides an overview of the devices used for each FPGA hardware implementation and corresponding software model pipeline architecture implementation profiled and reported in Section 5.1.

***Table 5.1:*** *Overview of devices used for each FPGA hardware implementation and corresponding software model pipeline architecture implementation.*

| Device Label | Manufacturer, Model, & Architecture | Clock Rate | CPU Cores | System RAM |
|---|---|---|---|---|
| FPGA | Altera Stratix V GS 5SGSMD5-K2F40C2N (FPGA) | 50 MHz | - | - |
| Core i7 | Intel Core i7 6700K (x86_64) | 4.00 GHz | 8 | 16 GB |
| Core 2 | Intel Core 2 Duo P8600 (x86_64) | 2.40 GHz | 2 | 16 GB |
| Atom | Intel Atom N570 (x86_64) | 1.66 GHz | 4 | 2 GB |
| ARMv7 | Broadcom BCM2836 - ARM Cortex-A7 (ARMv7) | 900 MHz | 4 | 1 GB |

Table 5.2 shows each `ct0.` pipeline architecture FPGA hardware implementation stage-count, and, latency $t_L$ with FPGA Master Clock Frequency `clk` of 50MHz. Table 5.3 shows FPGA resource utilisation for each `ct0.` pipeline architecture implementation compiled and synthesised for the Altera Stratix V GS 5SGSMD5 FPGA device; rows shown in grey were not synthesised due to DSP block resource requirements exceeding available device DSP block resources.

***Table 5.2:*** *Pipeline architecture* `ct0.` *FPGA hardware implementation stage-count and latency $t_L$ with master clock* `clk` *rate of 50MHz.*

| Pipeline `ct0.` | | | |
|---|---|---|---|
| $\bar{\boldsymbol{x}}$ Dimension ($d$) | Support Vectors ($k$) | Pipeline Stages | Pipeline Latency $t_L$ ($\mu$s) |
| 2 | 4 | 9 | 0.18 |
| | 8 | 11 | 0.22 |
| | 16 | 13 | 0.26 |
| | 32 | 15 | 0.30 |
| 4 | 8 | 11 | 0.22 |
| | 16 | 13 | 0.26 |
| | 32 | 15 | 0.30 |
| 8 | 16 | 14 | 0.28 |
| | 32 | 16 | 0.32 |

***Table 5.3:*** *Pipeline architecture* `ct0.` *FPGA resource utilisation of Altera Stratix V GS 5SGSMD5 FPGA implementation.*

| Pipeline `ct0.` | | | | | | | |
|---|---|---|---|---|---|---|---|
| $\bar{\boldsymbol{x}}$ Dimension ($d$) | Support Vectors ($k$) | LABs (of 17,260) | ALMs (of 172,600) | Combinational ALUTs | | Logic Registers (of 345,200) | DSP Blocks (of 1,590) |
| | | | | Logic | Route | | |
| 2 | 4 | 298 | 1,221 | 2,442 | 497 | 2,399 | 72 |
| | 8 | 1,105 | 4,714 | 9,360 | 1,944 | 9,492 | 272 |
| | 16 | 4,897 | 18,631 | 36,640 | 5,533 | 37,791 | 1,056 |
| | 32 | - | 55,327 | 56,073 | - | 110,651 | 2,048 |
| 4 | 8 | 1,153 | 4,714 | 9,360 | 1,897 | 9,491 | 308 |
| | 16 | 5,019 | 18,614 | 36,640 | 6,191 | 37,790 | 1,192 |
| | 32 | - | 55,328 | 56,072 | - | 110,653 | 3,104 |
| 8 | 16 | 5,814 | 20,529 | 39,224 | 8,033 | 42,687 | 1,464 |
| | 32 | - | 55,329 | 56,074 | - | 110,655 | 4,160 |

Table 5.4 shows each `ct0.` pipeline architecture FPGA hardware implementation and corresponding software model latency / execution time $t_L$ performance metrics. Figure 5.1 illustrates each `ct0.` pipeline architecture FPGA hardware implementation and corresponding software model latency / execution time $t_L$ performance metrics.

***Table 5.4:*** *Pipeline architecture* `ct0.` *FPGA hardware implementation and corresponding software model latency / execution time $t_L$ performance metrics.*

| Pipeline `ct0.` | | | | | | |
|---|---|---|---|---|---|---|
| $\bar{\boldsymbol{x}}$ Dimension ($d$) | Support Vectors ($k$) | Latency / Execution Time $t_L$ ($\mu$s) | | | | |
| | | FPGA Hardware | Software Model - Mean Execution Time | | | |
| | | | Core i7 | Core 2 | Atom | ARMv7 |
| 2 | 4 | 0.18 | 287.70 | 484.36 | 1,839.73 | 3,388.57 |
| | 8 | 0.22 | 582.89 | 476.48 | 1,872.66 | 3,418.53 |
| | 16 | 0.26 | 443.88 | 491.56 | 1,833.43 | 3,502.48 |
| | 32 | 0.30 | 473.90 | 510.58 | 2,020.39 | 3,622.26 |
| 4 | 8 | 0.22 | 698.00 | 484.01 | 1,868.89 | 3,426.55 |
| | 16 | 0.26 | 790.55 | 505.45 | 1,869.46 | 3,497.59 |
| | 32 | 0.30 | 705.75 | 528.13 | 1,937.39 | 3,663.93 |
| 8 | 16 | 0.28 | 899.89 | 495.09 | 1,946.31 | 3,524.37 |
| | 32 | 0.32 | 417.90 | 510.79 | 2,081.73 | 3,760.25 |

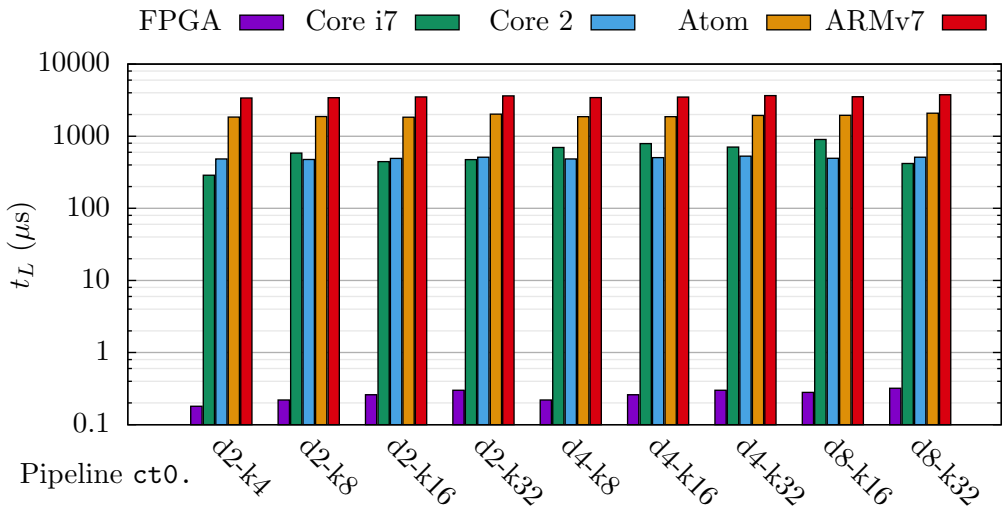



***Figure 5.1:*** *Pipeline architecture* `ct0.` *FPGA hardware implementation and corresponding software model latency / execution time $t_L$ performance metrics.*

Table 5.5 shows each `ct0.` pipeline architecture software model mean execution time $t_L$ performance metric's percentage standard deviation. Figure 5.2 illustrates each `ct0.` pipeline architecture software model mean execution time $t_L$ performance metric's percentage standard deviation.

***Table 5.5:*** *Pipeline architecture* `ct0.` *software model mean execution time $t_L$ performance metric's standard deviation (%).*

| Pipeline `ct0.` | | | | | |
|---|---|---|---|---|---|
| $\bar{\boldsymbol{x}}$ Dimension ($d$) | Support Vectors ($k$) | Software Model - Mean Execution Time Standard Deviation (%) | | | |
| | | Core i7 | Core 2 | Atom | ARMv7 |
| 2 | 4 | 1.04 | 1.40 | 1.78 | 0.28 |
| | 8 | 0.84 | 1.43 | 2.99 | 0.26 |
| | 16 | 1.14 | 1.46 | 1.65 | 0.32 |
| | 32 | 1.29 | 1.25 | 3.38 | 0.26 |
| 4 | 8 | 0.75 | 1.59 | 2.76 | 0.33 |
| | 16 | 0.77 | 1.23 | 2.31 | 0.28 |
| | 32 | 1.51 | 1.25 | 1.67 | 0.29 |
| 8 | 16 | 0.83 | 1.27 | 4.72 | 0.28 |
| | 32 | 3.01 | 1.36 | 5.36 | 0.26 |

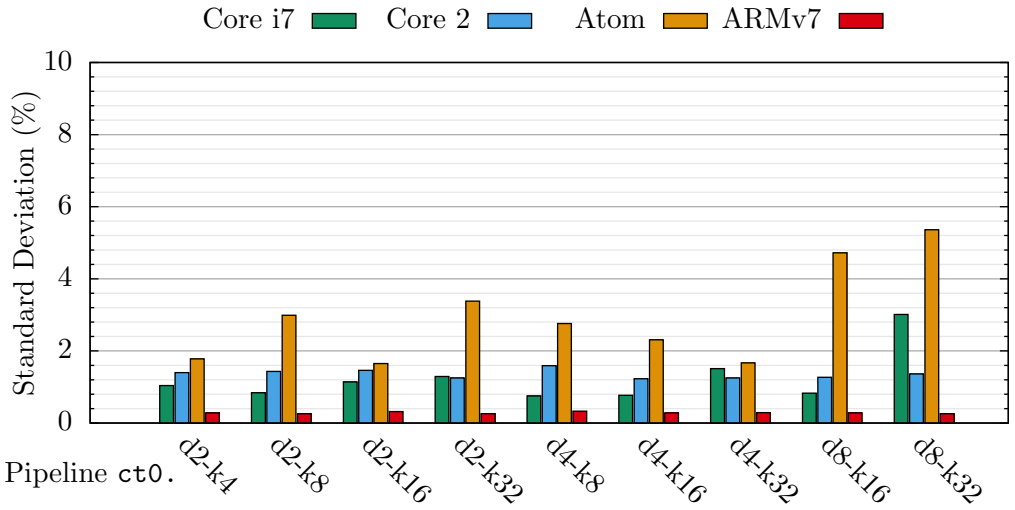



***Figure 5.2:*** *Pipeline architecture* `ct0.` *software model mean execution time $t_L$ performance metric's standard deviation (%).*

Table 5.6 shows each `ct0.` pipeline architecture FPGA hardware implementation and corresponding software model instructions-per-cycle performance metrics. Figure 5.3 illustrates each `ct0.` pipeline architecture FPGA hardware implementation and corresponding software model instructions-per-cycle performance metrics.

***Table 5.6:*** *Pipeline architecture* `ct0.` *FPGA hardware implementation and corresponding software model instructions-per-cycle performance metrics.*

| Pipeline `ct0.` | | | | | | | |
|---|---|---|---|---|---|---|---|
| $\bar{\boldsymbol{x}}$ Dimension ($d$) | Support Vectors ($k$) | Instructions-per-Cycle | | | | | |
| | | FPGA Hardware | | Software Model - Mean IPC | | | |
| | | DSP | Delays | Core i7 | Core 2 | Atom | ARMv7 |
| 2 | 4 | 136 | 28 | 0.74 | 0.67 | 0.30 | 0.28 |
| | 8 | 528 | 64 | 0.74 | 0.69 | 0.29 | 0.28 |
| | 16 | 2,080 | 144 | 0.75 | 0.69 | 0.31 | 0.28 |
| | 32 | 8,256 | 320 | 0.78 | 0.77 | 0.33 | 0.30 |
| 4 | 8 | 784 | 64 | 0.74 | 0.68 | 0.29 | 0.28 |
| | 16 | 3,104 | 144 | 0.76 | 0.71 | 0.31 | 0.28 |
| | 32 | 12,352 | 320 | 0.78 | 0.79 | 0.34 | 0.30 |
| 8 | 16 | 5,152 | 176 | 0.76 | 0.72 | 0.29 | 0.29 |
| | 32 | 20,544 | 384 | 0.78 | 0.83 | 0.31 | 0.31 |

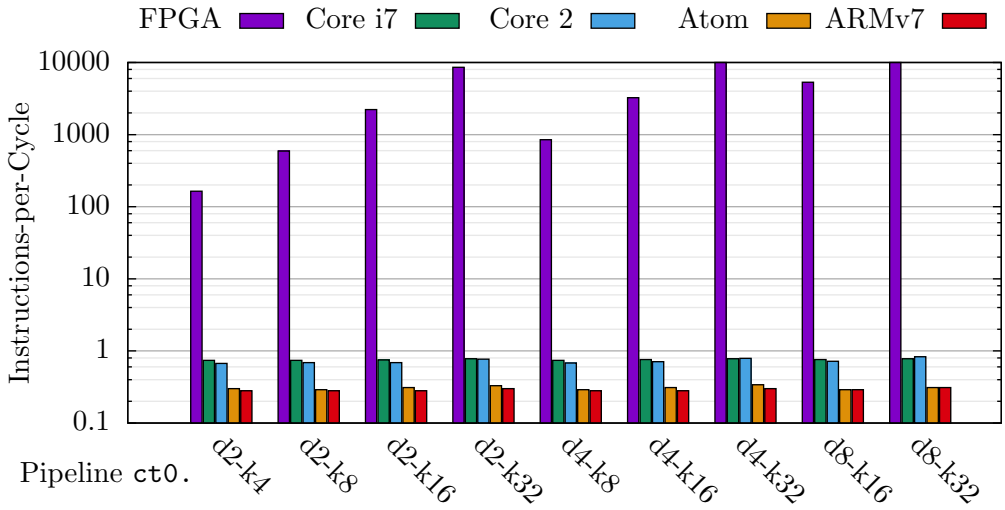



***Figure 5.3:*** *Pipeline architecture* `ct0.` *FPGA hardware implementation and corresponding software model instructions-per-cycle performance metrics.*

Table 5.7 shows each `ct0.` pipeline architecture software model mean instructions-per-cycle performance metric's percentage standard deviation. Figure 5.4 illustrates each `ct0.` pipeline architecture software model mean instructions-per-cycle performance metric's percentage standard deviation.

***Table 5.7:*** *Pipeline architecture* `ct0.` *software model mean instructions-per-cycle performance metric's standard deviation (%).*

| Pipeline `ct0.` | | | | | |
|---|---|---|---|---|---|
| $\bar{\boldsymbol{x}}$ Dimension ($d$) | Support Vectors ($k$) | Software Model Mean IPC Standard Deviation (%) | | | |
| | | Core i7 | Core 2 | Atom | ARMv7 |
| 2 | 4 | 1.00 | 0.93 | 2.05 | 0.35 |
| | 8 | 0.74 | 0.72 | 3.29 | 0.31 |
| | 16 | 1.00 | 0.81 | 1.55 | 0.36 |
| | 32 | 0.93 | 0.71 | 3.69 | 0.31 |
| 4 | 8 | 0.63 | 0.84 | 4.17 | 0.41 |
| | 16 | 0.61 | 0.75 | 2.11 | 0.34 |
| | 32 | 0.59 | 0.71 | 1.49 | 0.34 |
| 8 | 16 | 0.58 | 0.73 | 8.27 | 0.33 |
| | 32 | 0.78 | 0.64 | 9.47 | 0.30 |

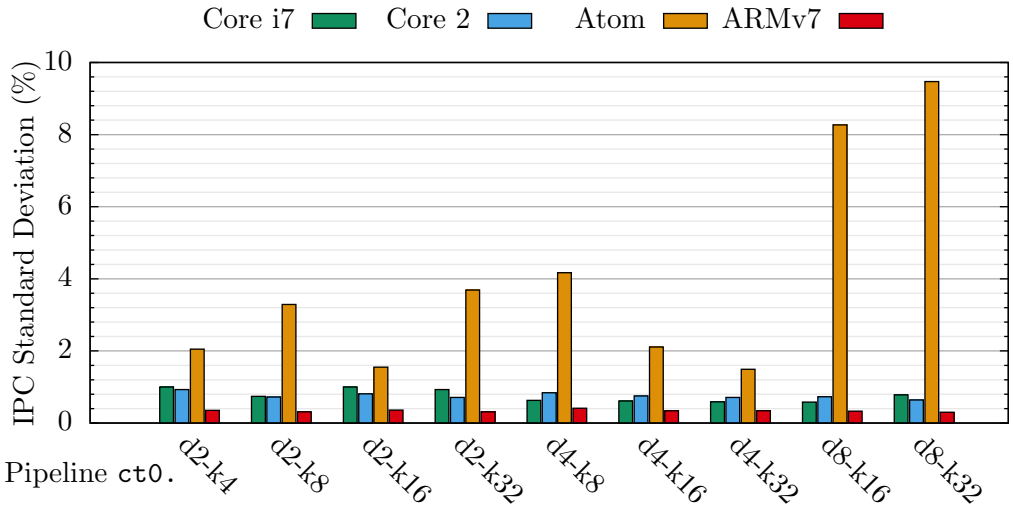



***Figure 5.4:*** *Pipeline architecture* `ct0.` *software model mean instructions-per-cycle performance metric's standard deviation (%).*

Table 5.8 shows each `ce0.` pipeline architecture FPGA hardware implementation stage-count, and, latency $t_L$ with FPGA Master Clock Frequency `clk` of 50MHz. Table 5.9 shows FPGA resource utilisation for each `ce0.` pipeline architecture implementation compiled and synthesised for the Altera Stratix V GS 5SGSMD5 FPGA device.

***Table 5.8:*** *Pipeline architecture* `ce0.` *FPGA hardware implementation stage-count and latency $t_L$ with master clock* `clk` *rate of 50MHz.*

| Pipeline `ce0.` | | | |
|---|---|---|---|
| $\bar{\boldsymbol{x}}$ Dimension ($d$) | Support Vectors ($k$) | Pipeline Stages | Pipeline Latency $t_L$ ($\mu$s) |
| 2 | 4 | 8 | 0.16 |
| | 8 | 9 | 0.18 |
| | 16 | 10 | 0.20 |
| | 32 | 11 | 0.22 |
| 4 | 8 | 9 | 0.18 |
| | 16 | 10 | 0.20 |
| | 32 | 11 | 0.22 |
| 8 | 16 | 11 | 0.22 |
| | 32 | 12 | 0.24 |

***Table 5.9:*** *Pipeline architecture* `ce0.` *FPGA resource utilisation of Altera Stratix V GS 5SGSMD5 FPGA implementation.*

| Pipeline `ce0.` | | | | | | | |
|---|---|---|---|---|---|---|---|
| $\bar{\boldsymbol{x}}$ Dimension ($d$) | Support Vectors ($k$) | LABs (of 17,260) | ALMs (of 172,600) | Combinational ALUTs | | Logic Registers (of 345,200) | DSP Blocks (of 1,590) |
| | | | | Logic | Route | | |
| 2 | 4 | 128 | 760 | 1,520 | 204 | 1,492 | 40 |
| | 8 | 305 | 1,548 | 3,095 | 364 | 2,982 | 80 |
| | 16 | 612 | 3,125 | 6,246 | 723 | 5,959 | 160 |
| | 32 | 1,234 | 6,275 | 12,549 | 1,297 | 11,911 | 320 |
| 4 | 8 | 305 | 1,548 | 3,095 | 387 | 2,981 | 104 |
| | 16 | 614 | 3,123 | 6,246 | 719 | 5,959 | 208 |
| | 32 | 1,212 | 6,275 | 12,549 | 1,339 | 11,911 | 416 |
| 8 | 16 | 862 | 3,724 | 7,158 | 1,482 | 7,688 | 304 |
| | 32 | 1,861 | 7,463 | 14,373 | 3,028 | 15,369 | 608 |

Table 5.10 shows each `ce0.` pipeline architecture FPGA hardware implementation and corresponding software model latency / execution time $t_L$ performance metrics. Figure 5.5 illustrates each `ce0.` pipeline architecture FPGA hardware implementation and corresponding software model latency / execution time $t_L$ performance metrics.

***Table 5.10:*** *Pipeline architecture* `ce0.` *FPGA hardware implementation and corresponding software model latency / execution time $t_L$ performance metrics.*

| Pipeline `ce0.` | | | | | | |
|---|---|---|---|---|---|---|
| $\bar{\boldsymbol{x}}$ Dimension ($d$) | Support Vectors ($k$) | Latency / Execution Time $t_L$ ($\mu$s) | | | | |
| | | FPGA Hardware | Software Model - Mean Execution Time | | | |
| | | | Core i7 | Core 2 | Atom | ARMv7 |
| 2 | 4 | 0.16 | 468.59 | 475.08 | 1,940.98 | 3,389.54 |
| | 8 | 0.18 | 557.74 | 493.41 | 1,943.65 | 3,415.09 |
| | 16 | 0.20 | 415.98 | 495.60 | 1,838.28 | 3,407.97 |
| | 32 | 0.22 | 431.59 | 481.88 | 1,851.31 | 3,434.48 |
| 4 | 8 | 0.18 | 697.77 | 490.11 | 1,876.49 | 3,416.29 |
| | 16 | 0.20 | 774.84 | 482.80 | 1,878.78 | 3,420.46 |
| | 32 | 0.22 | 780.33 | 474.03 | 1,897.28 | 3,456.02 |
| 8 | 16 | 0.22 | 873.84 | 496.59 | 1,820.47 | 3,449.01 |
| | 32 | 0.24 | 280.83 | 492.25 | 1,857.05 | 3,453.54 |

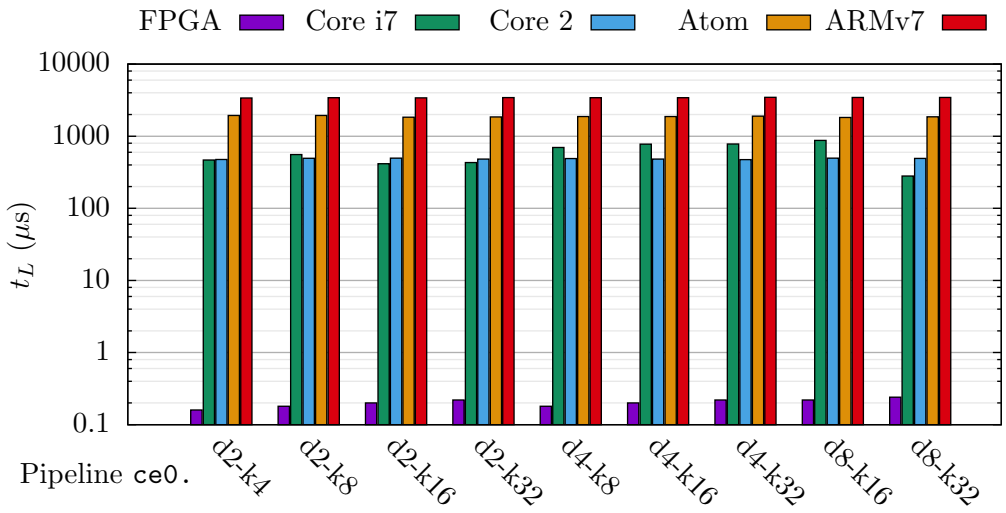



***Figure 5.5:*** *Pipeline architecture* `ce0.` *FPGA hardware implementation and corresponding software model latency / execution time $t_L$ performance metrics.*

Table 5.11 shows each `ce0.` pipeline architecture software model mean execution time $t_L$ performance metric's percentage standard deviation. Figure 5.6 illustrates each `ce0.` pipeline architecture software model mean execution time $t_L$ performance metric's percentage standard deviation.

***Table 5.11:*** *Pipeline architecture* `ce0.` *software model mean execution time $t_L$ performance metric's standard deviation (%).*

| Pipeline `ce0.` | | | | | |
|---|---|---|---|---|---|
| $\bar{\boldsymbol{x}}$ Dimension ($d$) | Support Vectors ($k$) | Software Model - Mean Execution Time Standard Deviation (%) | | | |
| | | Core i7 | Core 2 | Atom | ARMv7 |
| 2 | 4 | 3.97 | 1.44 | 5.54 | 0.30 |
| | 8 | 1.21 | 1.38 | 4.35 | 0.27 |
| | 16 | 1.67 | 1.60 | 1.76 | 0.28 |
| | 32 | 1.07 | 1.52 | 1.93 | 0.30 |
| 4 | 8 | 0.89 | 1.23 | 3.81 | 0.26 |
| | 16 | 0.76 | 1.18 | 3.44 | 0.26 |
| | 32 | 0.91 | 1.40 | 3.27 | 0.30 |
| 8 | 16 | 0.83 | 1.80 | 2.15 | 0.32 |
| | 32 | 1.07 | 1.36 | 2.75 | 0.29 |

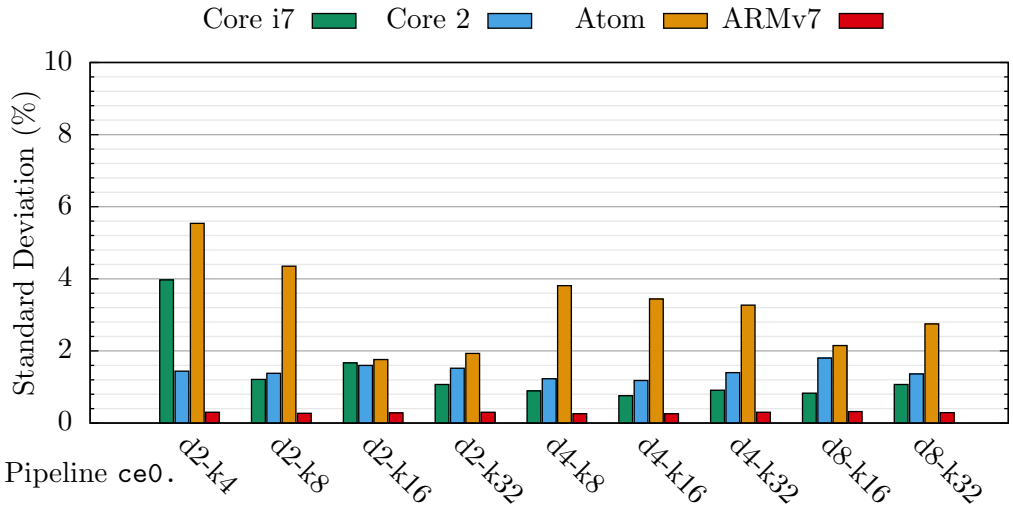



***Figure 5.6:*** *Pipeline architecture* `ce0.` *software model mean execution time $t_L$ performance metric's standard deviation (%).*

Table 5.12 shows each `ce0.` pipeline architecture FPGA hardware implementation and corresponding software model instructions-per-cycle performance metrics. Figure 5.7 illustrates each `ce0.` pipeline architecture FPGA hardware implementation and corresponding software model instructions-per-cycle performance metrics.

***Table 5.12:*** *Pipeline architecture* `ce0.` *FPGA hardware implementation and corresponding software model instructions-per-cycle performance metrics.*

| Pipeline `ce0.` | | | | | | | |
|---|---|---|---|---|---|---|---|
| $\bar{\boldsymbol{x}}$ Dimension ($d$) | Support Vectors ($k$) | Instructions-per-Cycle | | | | | |
| | | FPGA Hardware | | Software Model - Mean IPC | | | |
| | | DSP | Delays | Core i7 | Core 2 | Atom | ARMv7 |
| 2 | 4 | 88 | 17 | 0.74 | 0.67 | 0.27 | 0.28 |
| | 8 | 176 | 33 | 0.74 | 0.67 | 0.27 | 0.28 |
| | 16 | 352 | 65 | 0.74 | 0.68 | 0.30 | 0.28 |
| | 32 | 704 | 129 | 0.75 | 0.68 | 0.30 | 0.28 |
| 4 | 8 | 272 | 33 | 0.74 | 0.68 | 0.29 | 0.28 |
| | 16 | 544 | 65 | 0.74 | 0.68 | 0.29 | 0.28 |
| | 32 | 1,088 | 129 | 0.74 | 0.69 | 0.29 | 0.28 |
| 8 | 16 | 928 | 97 | 0.75 | 0.67 | 0.31 | 0.28 |
| | 32 | 1,856 | 193 | 0.74 | 0.69 | 0.30 | 0.28 |

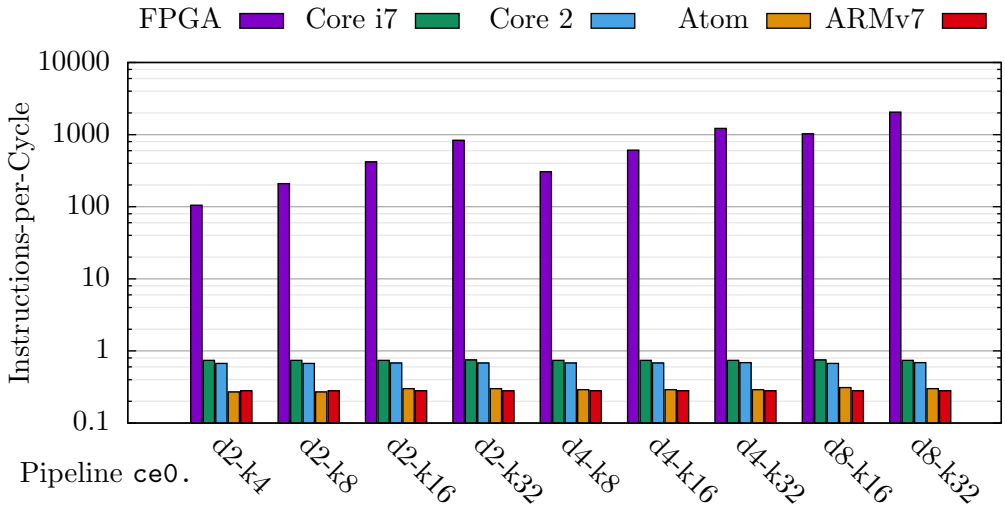



***Figure 5.7:*** *Pipeline architecture* `ce0.` *FPGA hardware implementation and corresponding software model instructions-per-cycle performance metrics.*

Table 5.13 shows each `ce0.` pipeline architecture software model mean instructions-per-cycle performance metric's percentage standard deviation. Figure 5.8 illustrates each `ce0.` pipeline architecture software model mean instructions-per-cycle performance metric's percentage standard deviation.

***Table 5.13:*** *Pipeline architecture* `ce0.` *software model mean instructions-per-cycle performance metric's standard deviation (%).*

| Pipeline `ce0.` | | | | | |
|---|---|---|---|---|---|
| $\bar{\boldsymbol{x}}$ Dimension ($d$) | Support Vectors ($k$) | Software Model Mean IPC Standard Deviation (%) | | | |
| | | Core i7 | Core 2 | Atom | ARMv7 |
| 2 | 4 | 0.82 | 0.84 | 9.03 | 0.36 |
| | 8 | 0.84 | 0.79 | 7.56 | 0.32 |
| | 16 | 1.03 | 0.79 | 2.03 | 0.34 |
| | 32 | 1.04 | 0.82 | 1.58 | 0.36 |
| 4 | 8 | 0.71 | 0.67 | 5.24 | 0.31 |
| | 16 | 0.61 | 0.79 | 3.30 | 0.31 |
| | 32 | 0.76 | 0.73 | 3.76 | 0.36 |
| 8 | 16 | 0.67 | 0.98 | 2.19 | 0.38 |
| | 32 | 1.03 | 0.70 | 2.87 | 0.35 |

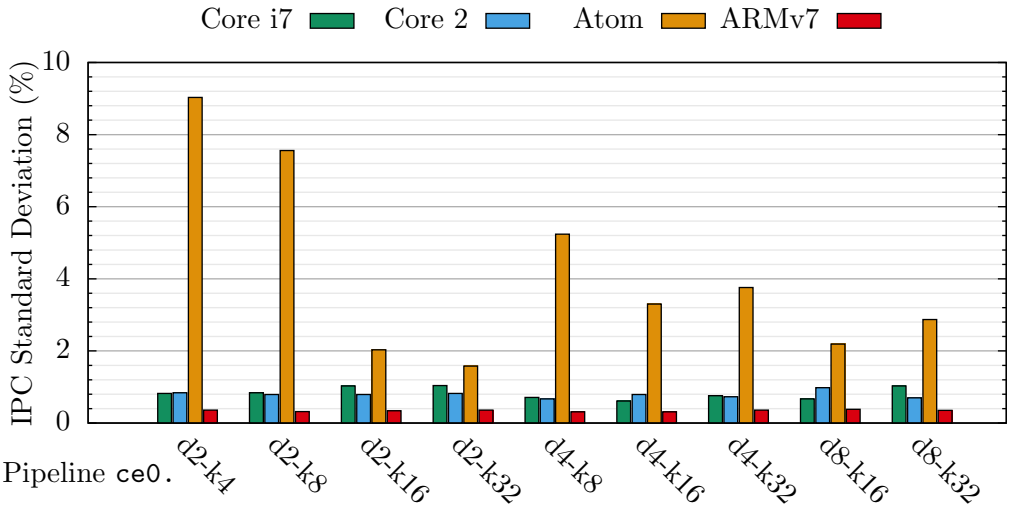



***Figure 5.8:*** *Pipeline architecture* `ce0.` *software model mean instructions-per-cycle performance metric's standard deviation (%).*

Table 5.14 shows each `rt0.` pipeline architecture FPGA hardware implementation stage-count, and, latency $t_L$ with FPGA Master Clock Frequency `clk` of 50MHz. Table 5.15 shows FPGA resource utilisation for each `rt0.` pipeline architecture implementation compiled and synthesised for the Altera Stratix V GS 5SGSMD5 FPGA device; rows shown in grey were not synthesised due to DSP block resource requirements exceeding available device DSP block resources.

***Table 5.14:*** *Pipeline architecture* `rt0.` *FPGA hardware implementation stage-count and latency $t_L$ with master clock* `clk` *rate of 50MHz.*

| Pipeline `rt0.` | | | |
|---|---|---|---|
| $\bar{\boldsymbol{x}}$ Dimension ($d$) | Support Vectors ($k$) | Pipeline Stages | Pipeline Latency $t_L$ ($\mu$s) |
| 2 | 4 | 10 | 0.20 |
| | 8 | 12 | 0.24 |
| | 16 | 14 | 0.28 |
| | 32 | 16 | 0.32 |
| 4 | 8 | 12 | 0.24 |
| | 16 | 14 | 0.28 |
| | 32 | 16 | 0.32 |
| 8 | 16 | 15 | 0.30 |
| | 32 | 17 | 0.34 |

***Table 5.15:*** *Pipeline architecture* `rt0.` *FPGA resource utilisation of Altera Stratix V GS 5SGSMD5 FPGA implementation.*

| Pipeline `rt0.` | | | | | | | |
|---|---|---|---|---|---|---|---|
| $\bar{\boldsymbol{x}}$ Dimension ($d$) | Support Vectors ($k$) | LABs (of 17,260) | ALMs (of 172,600) | Combinational ALUTs | | Logic Registers (of 345,200) | DSP Blocks (of 1,590) |
| | | | | Logic | Route | | |
| 2 | 4 | 219 | 1,071 | 2,087 | 316 | 2,192 | 56 |
| | 8 | 809 | 4,045 | 8,008 | 850 | 8,161 | 208 |
| | 16 | 3,461 | 15,651 | 31,205 | 3,921 | 31,358 | 800 |
| | 32 | - | 71,664 | 57,373 | - | 143,327 | 1,873 |
| 4 | 8 | 813 | 4,044 | 8,008 | 909 | 8,161 | 244 |
| | 16 | 3,618 | 15,644 | 31,205 | 4,424 | 31,360 | 936 |
| | 32 | - | 71,672 | 57,388 | - | 143,342 | 2,929 |
| 8 | 16 | 4,639 | 17,547 | 33,783 | 6,694 | 36,249 | 1,208 |
| | 32 | - | 71,673 | 57,390 | - | 143,345 | 3,985 |

Table 5.16 shows each `rt0.` pipeline architecture FPGA hardware implementation and corresponding software model latency / execution time $t_L$ performance metrics. Figure 5.9 illustrates each `rt0.` pipeline architecture FPGA hardware implementation and corresponding software model latency / execution time $t_L$ performance metrics.

***Table 5.16:*** *Pipeline architecture* `rt0.` *FPGA hardware implementation and corresponding software model latency / execution time $t_L$ performance metrics.*

| Pipeline `rt0.` | | | | | | |
|---|---|---|---|---|---|---|
| $\bar{\boldsymbol{x}}$ Dimension ($d$) | Support Vectors ($k$) | Latency / Execution Time $t_L$ ($\mu$s) | | | | |
| | | FPGA Hardware | Software Model - Mean Execution Time | | | |
| | | | Core i7 | Core 2 | Atom | ARMv7 |
| 2 | 4 | 0.20 | 358.84 | 485.00 | 1,767.10 | 3,415.29 |
| | 8 | 0.24 | 615.85 | 478.83 | 1,831.73 | 3,422.07 |
| | 16 | 0.28 | 441.32 | 475.66 | 1,893.29 | 3,492.58 |
| | 32 | 0.32 | 466.77 | 510.11 | 1,908.30 | 3,639.97 |
| 4 | 8 | 0.24 | 811.61 | 486.31 | 1,790.73 | 3,443.56 |
| | 16 | 0.28 | 789.19 | 489.94 | 1,849.89 | 3,502.98 |
| | 32 | 0.32 | 760.88 | 522.30 | 2,015.19 | 3,654.13 |
| 8 | 16 | 0.30 | 258.32 | 500.69 | 1,939.84 | 3,514.52 |
| | 32 | 0.34 | 605.40 | 533.47 | 2,076.28 | 3,749.97 |

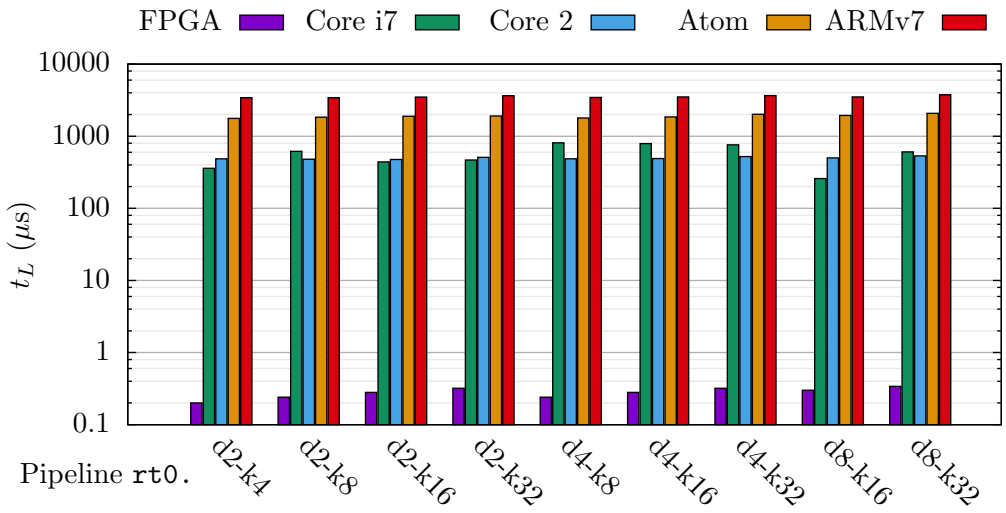



***Figure 5.9:*** *Pipeline architecture* `rt0.` *FPGA hardware implementation and corresponding software model latency / execution time $t_L$ performance metrics.*

Table 5.17 shows each `rt0.` pipeline architecture software model mean execution time $t_L$ performance metric's percentage standard deviation. Figure 5.10 illustrates each `rt0.` pipeline architecture software model mean execution time $t_L$ performance metric's percentage standard deviation.

***Table 5.17:*** *Pipeline architecture* `rt0.` *software model mean execution time $t_L$ performance metric's standard deviation (%).*

| Pipeline `rt0.` | | | | | |
|---|---|---|---|---|---|
| $\bar{\boldsymbol{x}}$ Dimension ($d$) | Support Vectors ($k$) | Software Model - Execution Time Standard Deviation (%) | | | |
| | | Core i7 | Core 2 | Atom | ARMv7 |
| 2 | 4 | 1.45 | 1.28 | 1.36 | 0.25 |
| | 8 | 1.47 | 1.34 | 1.71 | 0.30 |
| | 16 | 0.81 | 1.11 | 4.08 | 0.27 |
| | 32 | 0.76 | 1.51 | 1.73 | 0.31 |
| 4 | 8 | 1.24 | 1.31 | 1.36 | 0.30 |
| | 16 | 0.87 | 1.78 | 2.17 | 0.28 |
| | 32 | 0.76 | 1.18 | 5.12 | 0.26 |
| 8 | 16 | 0.95 | 1.27 | 4.63 | 0.28 |
| | 32 | 0.75 | 1.12 | 4.07 | 0.26 |

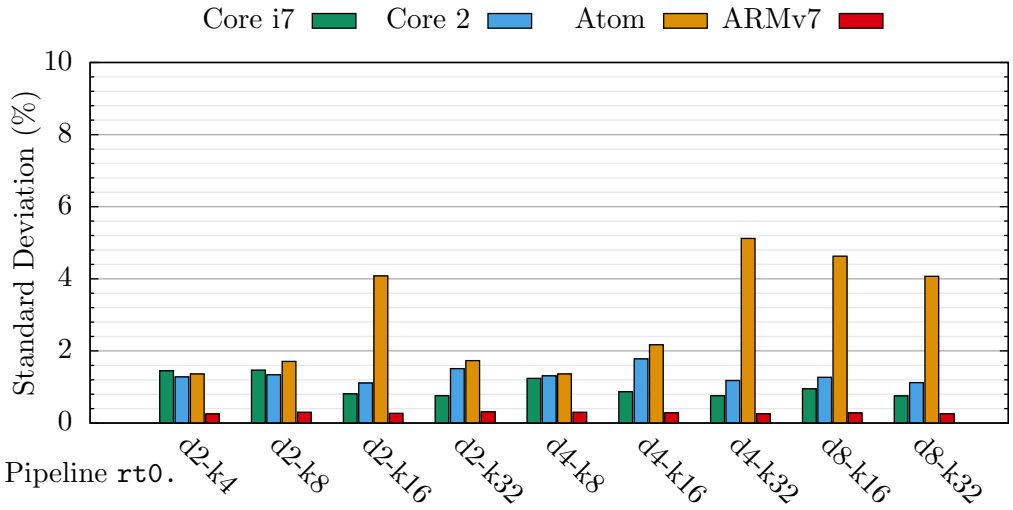



***Figure 5.10:*** *Pipeline architecture* `rt0.` *software model mean execution time $t_L$ performance metric's standard deviation (%).*

Table 5.18 shows each `rt0.` pipeline architecture FPGA hardware implementation and corresponding software model instructions-per-cycle performance metrics. Figure 5.11 illustrates each `rt0.` pipeline architecture FPGA hardware implementation and corresponding software model instructions-per-cycle performance metrics.

***Table 5.18:*** *Pipeline architecture* `rt0.` *FPGA hardware implementation and corresponding software model instructions-per-cycle performance metrics.*

| Pipeline `rt0.` | | | | | | | |
|---|---|---|---|---|---|---|---|
| $\bar{\boldsymbol{x}}$ Dimension ($d$) | Support Vectors ($k$) | Instructions-per-Cycle | | | | | |
| | | FPGA Hardware | | Software Model - Mean IPC | | | |
| | | DSP | Delays | Core i7 | Core 2 | Atom | ARMv7 |
| 2 | 4 | 149 | 36 | 0.74 | 0.68 | 0.31 | 0.28 |
| | 8 | 553 | 70 | 0.74 | 0.67 | 0.31 | 0.28 |
| | 16 | 2,129 | 136 | 0.75 | 0.70 | 0.29 | 0.28 |
| | 32 | 8,353 | 266 | 0.78 | 0.77 | 0.33 | 0.30 |
| 4 | 8 | 809 | 70 | 0.74 | 0.68 | 0.31 | 0.28 |
| | 16 | 3,153 | 136 | 0.75 | 0.71 | 0.31 | 0.28 |
| | 32 | 12.449 | 266 | 0.78 | 0.79 | 0.32 | 0.30 |
| 8 | 16 | 5,201 | 184 | 0.75 | 0.72 | 0.29 | 0.29 |
| | 32 | 20,641 | 362 | 0.78 | 0.83 | 0.33 | 0.31 |

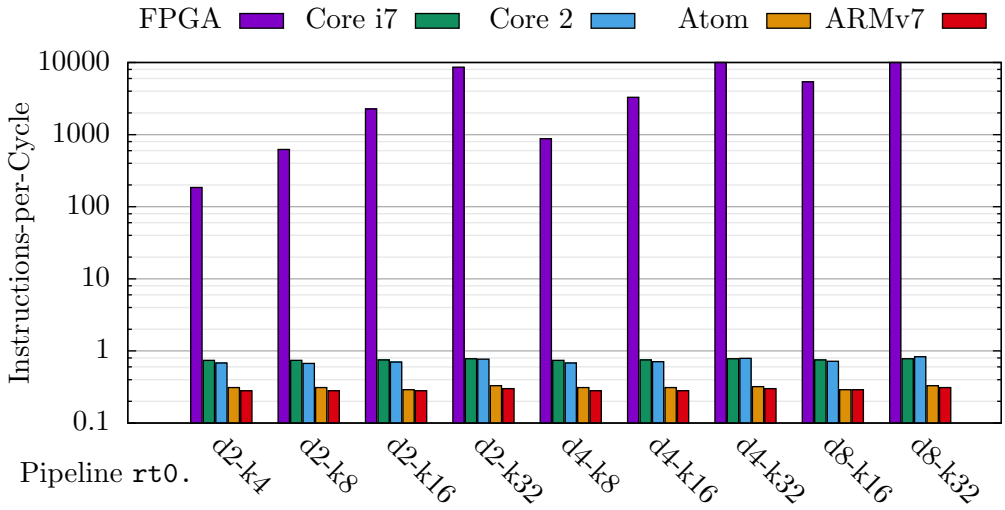



***Figure 5.11:*** *Pipeline architecture* `rt0.` *FPGA hardware implementation and corresponding software model instructions-per-cycle performance metrics.*

Table 5.19 shows each `rt0.` pipeline architecture software model mean instructions-per-cycle performance metric's percentage standard deviation. Figure 5.12 illustrates each `rt0.` pipeline architecture software model mean instructions-per-cycle performance metric's percentage standard deviation.

***Table 5.19:*** *Pipeline architecture* `rt0.` *software model mean instructions-per-cycle performance metric's standard deviation (%).*

| Pipeline `rt0.` | | | | | |
|---|---|---|---|---|---|
| $\bar{\boldsymbol{x}}$ Dimension ($d$) | Support Vectors ($k$) | Software Model Mean IPC Standard Deviation (%) | | | |
| | | Core i7 | Core 2 | Atom | ARMv7 |
| 2 | 4 | 0.90 | 0.77 | 1.25 | 0.30 |
| | 8 | 0.70 | 0.81 | 1.57 | 0.37 |
| | 16 | 0.70 | 0.66 | 5.75 | 0.32 |
| | 32 | 0.68 | 0.81 | 1.61 | 0.36 |
| 4 | 8 | 0.63 | 0.85 | 1.27 | 0.36 |
| | 16 | 0.67 | 0.84 | 2.58 | 0.34 |
| | 32 | 0.58 | 0.70 | 8.56 | 0.31 |
| 8 | 16 | 0.98 | 0.76 | 9.66 | 0.35 |
| | 32 | 0.55 | 0.68 | 5.59 | 0.30 |

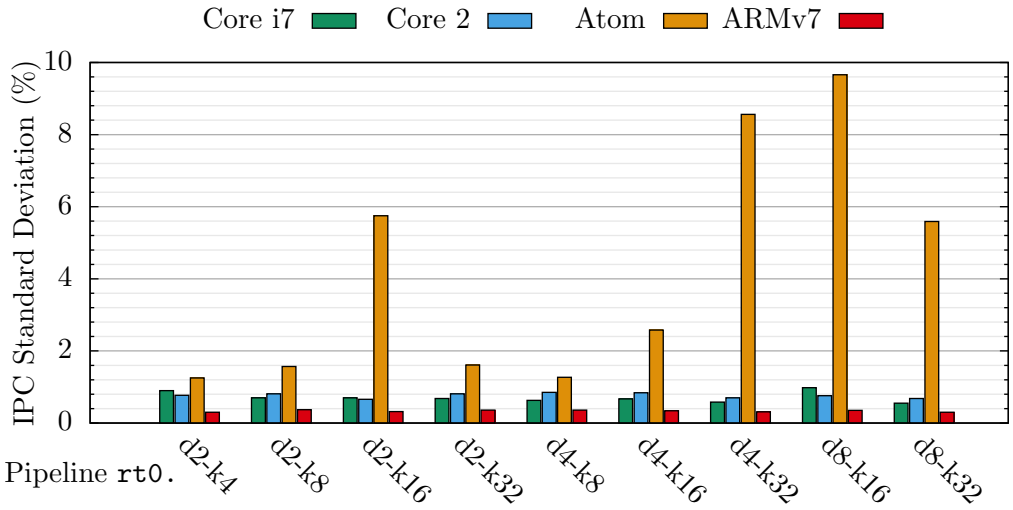



***Figure 5.12:*** *Pipeline architecture* `rt0.` *software model mean instructions-per-cycle performance metric's standard deviation (%).*

Table 5.20 shows each `re0.` pipeline architecture FPGA hardware implementation stage-count, and, latency $t_L$ with FPGA Master Clock Frequency `clk` of 50MHz. Table 5.21 shows FPGA resource utilisation for each `re0.` pipeline architecture implementation compiled and synthesised for the Altera Stratix V GS 5SGSMD5 FPGA device; rows shown in grey were not synthesised due to DSP block resource requirements exceeding available device DSP block resources.

***Table 5.20:*** *Pipeline architecture* `re0.` *FPGA hardware implementation stage-count and latency $t_L$ with master clock* `clk` *rate of 50MHz.*

| Pipeline `re0.` | | | |
|---|---|---|---|
| $\bar{\boldsymbol{x}}$ Dimension ($d$) | Support Vectors ($k$) | Pipeline Stages | Pipeline Latency $t_L$ ($\mu$s) |
| 2 | 4 | 10 | 0.20 |
| | 8 | 12 | 0.24 |
| | 16 | 14 | 0.28 |
| | 32 | 16 | 0.32 |
| 4 | 8 | 12 | 0.24 |
| | 16 | 14 | 0.28 |
| | 32 | 16 | 0.32 |
| 8 | 16 | 15 | 0.30 |
| | 32 | 17 | 0.34 |

***Table 5.21:*** *Pipeline architecture* `re0.` *FPGA resource utilisation of Altera Stratix V GS 5SGSMD5 FPGA implementation.*

| Pipeline `re0.` | | | | | | | |
|---|---|---|---|---|---|---|---|
| $\bar{\boldsymbol{x}}$ Dimension ($d$) | Support Vectors ($k$) | LABs (of 17,260) | ALMs (of 172,600) | Combinational ALUTs | | Logic Registers (of 345,200) | DSP Blocks (of 1,590) |
| | | | | Logic | Route | | |
| 2 | 4 | 269 | 1,485 | 2,867 | 338 | 3,041 | 54 |
| | 8 | 947 | 5,389 | 10,518 | 914 | 10,815 | 188 |
| | 16 | 4,231 | 20,748 | 40,032 | 4,374 | 40,529 | 696 |
| | 32 | - | 84,022 | 96,259 | - | 168,042 | 1,772 |
| 4 | 8 | 973 | 5,386 | 10,518 | 1,031 | 10,815 | 232 |
| | 16 | 4,306 | 21,064 | 40,032 | 4,536 | 40,529 | 848 |
| | 32 | - | 84,022 | 96,259 | - | 168,042 | 2,892 |
| 8 | 16 | 5,217 | 23,005 | 42,921 | 7,490 | 46,004 | 1,152 |
| | 32 | - | 84,023 | 96,261 | - | 168,045 | 4,012 |

Table 5.22 shows each `re0.` pipeline architecture FPGA hardware implementation and corresponding software model latency / execution time $t_L$ performance metrics. Figure 5.13 illustrates each `re0.` pipeline architecture FPGA hardware implementation and corresponding software model latency / execution time $t_L$ performance metrics.

***Table 5.22:*** *Pipeline architecture* `re0.` *FPGA hardware implementation and corresponding software model latency / execution time $t_L$ performance metrics.*

| Pipeline `re0.` | | | | | | |
|---|---|---|---|---|---|---|
| $\bar{\boldsymbol{x}}$ Dimension ($d$) | Support Vectors ($k$) | Latency / Execution Time $t_L$ ($\mu$s) | | | | |
| | | FPGA Hardware | Software Model - Mean Execution Time | | | |
| | | | Core i7 | Core 2 | Atom | ARMv7 |
| 2 | 4 | 0.20 | 290.61 | 485.82 | 1,812.11 | 3,387.95 |
| | 8 | 0.24 | 588.58 | 476.98 | 1,886.88 | 3,438.00 |
| | 16 | 0.28 | 441.73 | 500.30 | 1,915.64 | 3,487.79 |
| | 32 | 0.32 | 471.09 | 501.00 | 1,962.67 | 3,647.48 |
| 4 | 8 | 0.24 | 701.52 | 510.49 | 1,757.86 | 3,463.73 |
| | 16 | 0.28 | 795.90 | 497.51 | 1,904.74 | 3,487.99 |
| | 32 | 0.32 | 764.98 | 521.76 | 2,009.64 | 3,654.92 |
| 8 | 16 | 0.30 | 298.66 | 494.90 | 1,872.30 | 3,537.62 |
| | 32 | 0.34 | 608.03 | 520.21 | 1,986.22 | 3,764.51 |

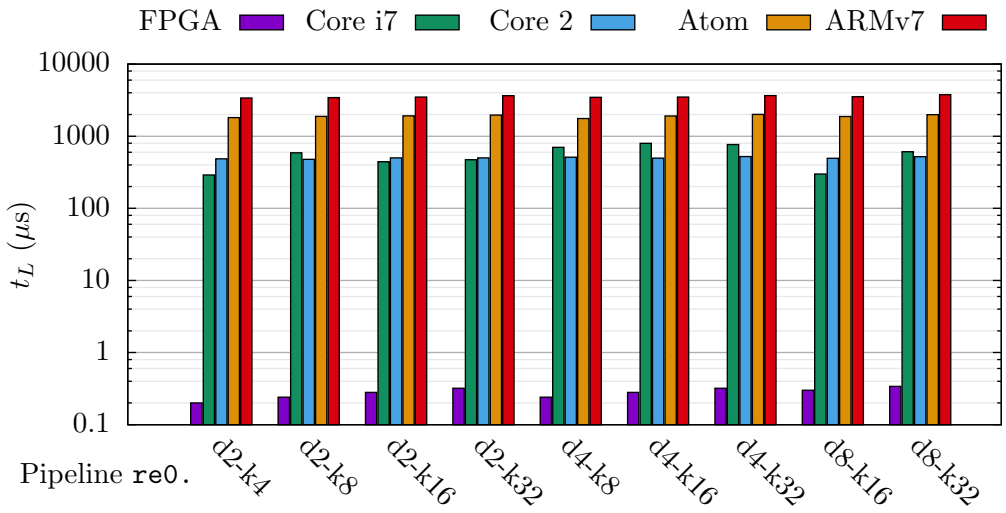



***Figure 5.13:*** *Pipeline architecture* `re0.` *FPGA hardware implementation and corresponding software model latency / execution time $t_L$ performance metrics.*

Table 5.23 shows each `re0.` pipeline architecture software model mean execution time $t_L$ performance metric's percentage standard deviation. Figure 5.14 illustrates each `re0.` pipeline architecture software model mean execution time $t_L$ performance metric's percentage standard deviation.

***Table 5.23:*** *Pipeline architecture* `re0.` *software model mean execution time $t_L$ performance metric's standard deviation (%).*

| Pipeline `re0.` | | | | | |
|---|---|---|---|---|---|
| $\bar{\boldsymbol{x}}$ Dimension ($d$) | Support Vectors ($k$) | Software Model - Mean Execution Time Standard Deviation (%) | | | |
| | | Core i7 | Core 2 | Atom | ARMv7 |
| 2 | 4 | 0.97 | 1.14 | 1.72 | 0.29 |
| | 8 | 0.87 | 1.47 | 4.42 | 0.28 |
| | 16 | 0.90 | 1.22 | 3.20 | 0.27 |
| | 32 | 0.93 | 1.18 | 2.52 | 0.26 |
| 4 | 8 | 0.80 | 1.03 | 1.33 | 0.27 |
| | 16 | 0.92 | 1.12 | 2.70 | 0.25 |
| | 32 | 0.82 | 1.32 | 4.28 | 0.24 |
| 8 | 16 | 4.29 | 1.00 | 2.41 | 0.26 |
| | 32 | 0.72 | 1.49 | 2.57 | 0.24 |

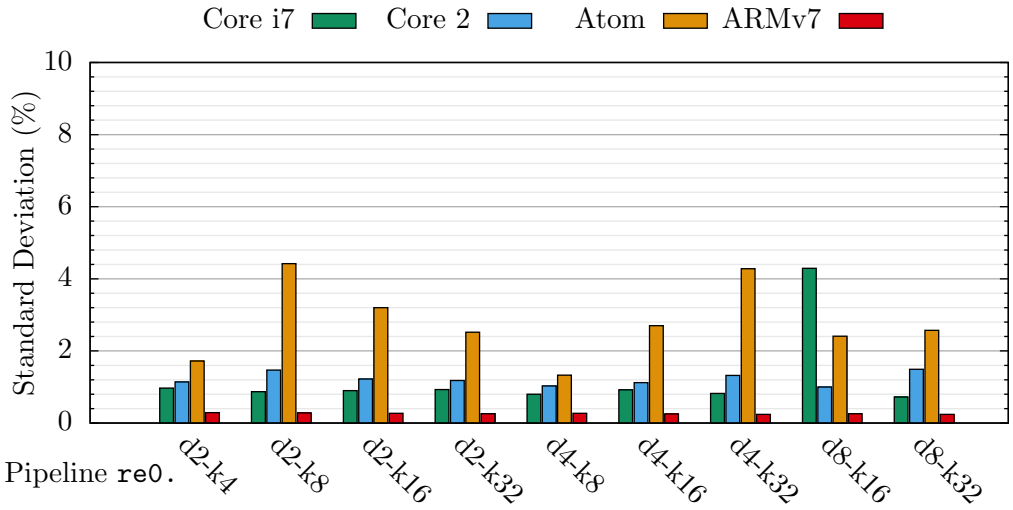



***Figure 5.14:*** *Pipeline architecture* `re0.` *software model mean execution time $t_L$ performance metric's standard deviation (%).*

Table 5.24 shows each `re0.` pipeline architecture FPGA hardware implementation and corresponding software model instructions-per-cycle performance metrics. Figure 5.15 illustrates each `re0.` pipeline architecture FPGA hardware implementation and corresponding software model instructions-per-cycle performance metrics.

***Table 5.24:*** *Pipeline architecture* `re0.` *FPGA hardware implementation and corresponding software model instructions-per-cycle performance metrics.*

| Pipeline `re0.` | | | | | | | |
|---|---|---|---|---|---|---|---|
| $\bar{\boldsymbol{x}}$ Dimension ($d$) | Support Vectors ($k$) | Instructions-per-Cycle | | | | | |
| | | FPGA Hardware | | Software Model - Mean IPC | | | |
| | | DSP | Delays | Core i7 | Core 2 | Atom | ARMv7 |
| 2 | 4 | 160 | 38 | 0.73 | 0.68 | 0.30 | 0.28 |
| | 8 | 576 | 75 | 0.74 | 0.68 | 0.29 | 0.28 |
| | 16 | 2,176 | 148 | 0.75 | 0.68 | 0.30 | 0.29 |
| | 32 | 8,448 | 293 | 0.78 | 0.77 | 0.33 | 0.30 |
| 4 | 8 | 864 | 75 | 0.74 | 0.68 | 0.31 | 0.28 |
| | 16 | 3,264 | 148 | 0.75 | 0.71 | 0.29 | 0.29 |
| | 32 | 12,672 | 293 | 0.78 | 0.79 | 0.31 | 0.30 |
| 8 | 16 | 5,568 | 196 | 0.75 | 0.71 | 0.30 | 0.29 |
| | 32 | 21,120 | 389 | 0.78 | 0.84 | 0.34 | 0.31 |

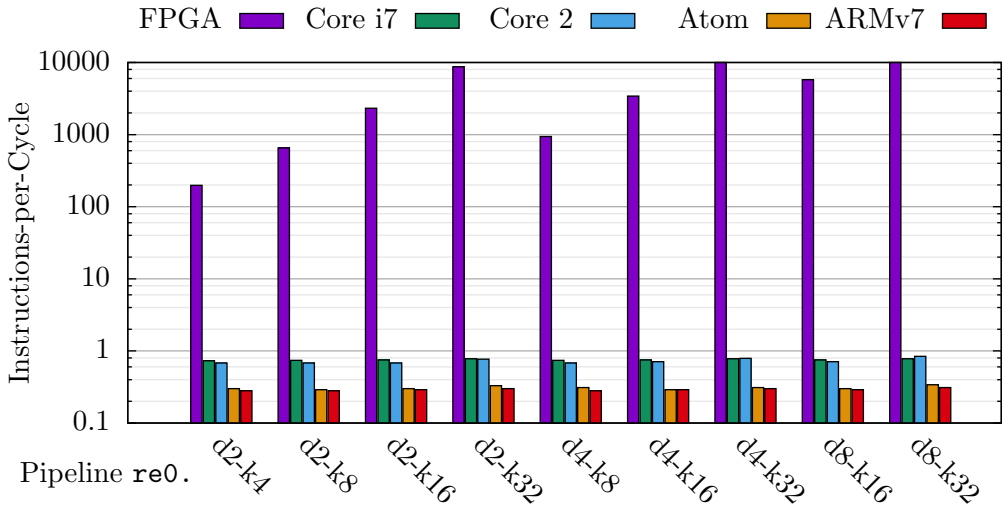



***Figure 5.15:*** *Pipeline architecture* `re0.` *FPGA hardware implementation and corresponding software model instructions-per-cycle performance metrics.*

Table 5.25 shows each `re0.` pipeline architecture software model mean instructions-per-cycle performance metric's percentage standard deviation. Figure 5.16 illustrates each `re0.` pipeline architecture software model mean instructions-per-cycle performance metric's percentage standard deviation.

***Table 5.25:*** *Pipeline architecture* `re0.` *software model mean instructions-per-cycle performance metric's standard deviation (%).*

| Pipeline `re0.` | | | | | |
|---|---|---|---|---|---|
| $\bar{\boldsymbol{x}}$ Dimension ($d$) | Support Vectors ($k$) | Software Model Mean IPC Standard Deviation (%) | | | |
| | | Core i7 | Core 2 | Atom | ARMv7 |
| 2 | 4 | 1.00 | 0.72 | 1.52 | 0.35 |
| | 8 | 0.72 | 0.77 | 5.57 | 0.35 |
| | 16 | 0.87 | 1.21 | 3.65 | 0.31 |
| | 32 | 0.76 | 0.73 | 2.38 | 0.30 |
| 4 | 8 | 0.71 | 0.75 | 1.39 | 0.32 |
| | 16 | 0.71 | 0.71 | 4.68 | 0.30 |
| | 32 | 0.61 | 0.71 | 9.68 | 0.28 |
| 8 | 16 | 0.99 | 0.84 | 4.08 | 0.31 |
| | 32 | 0.57 | 0.64 | 3.49 | 0.28 |

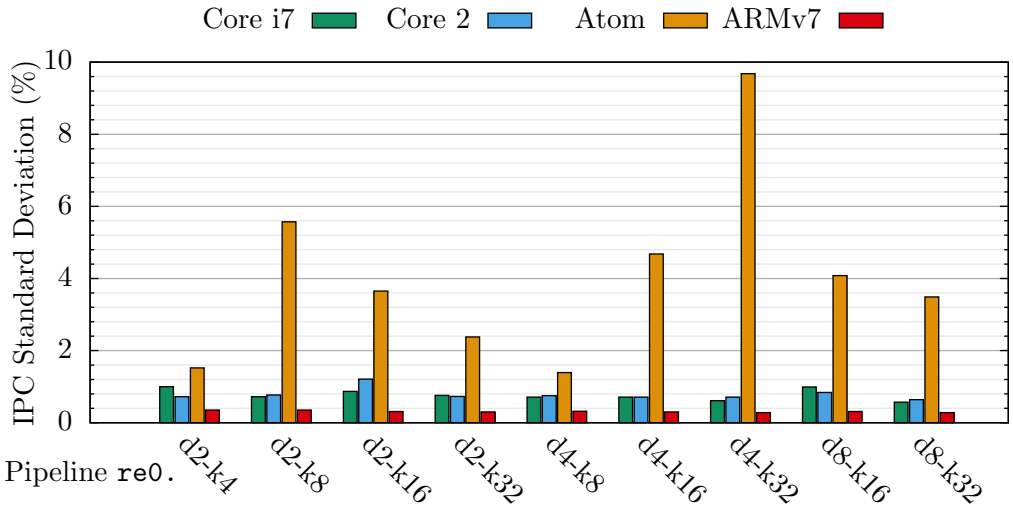



***Figure 5.16:*** *Pipeline architecture* `re0.` *software model mean instructions-per-cycle performance metric's standard deviation (%).*

## 5.2 Electrical Results

Average power consumption was found by calculating the product of the applied DC source voltage and measured DC current flowing into the Altera Stratix V GS 5SGSMD5 FPGA DSP development board running each DSP pipeline implementation with pipeline enable `en` rate of 20 kHz, or 50 MHz, applied to the pipeline circuit.

Idle-state power consumption of an Altera Stratix V GS 5SGSMD5 FPGA DSP develop-

ment board, running a trivial clock and synchronous-reset process hardware implementation, was calculated to be 10.75 W.

Table 5.26 shows the average power consumption of each Altera Stratix V GS 5SGSMD5 FPGA DSP development board `ct0.` pipeline architecture implementation with pipeline enable `en` rate of 20 kHz and 50 MHz. Figure 5.17 illustrates the average power consumption of each Altera Stratix V GS 5SGSMD5 FPGA DSP development board `ct0.` pipeline architecture implementation with pipeline enable `en` rate of 20 kHz and 50 MHz.

***Table 5.26:*** *Pipeline architecture* `ct0.` *average power consumption of each Altera Stratix V GS 5SGSMD5 FPGA DSP development board implementation with pipeline enable* `en` *rate of 20 kHz and 50 MHz.*

| Pipeline `ct0.` | | | |
|---|---|---|---|
| $\bar{\boldsymbol{x}}$ Dimension ($d$) | Support Vectors ($k$) | Power (W) @ pipeline `en` rate: | |
| | | 20 kHz | 50 MHz |
| 2 | 4 | 10.68 | 11.19 |
| | 8 | 11.02 | 11.21 |
| | 16 | 11.10 | 11.48 |
| | 32 | - | - |
| 4 | 8 | 10.91 | 10.89 |
| | 16 | 10.86 | 11.28 |
| | 32 | - | - |
| 8 | 16 | 10.99 | 11.31 |
| | 32 | - | - |

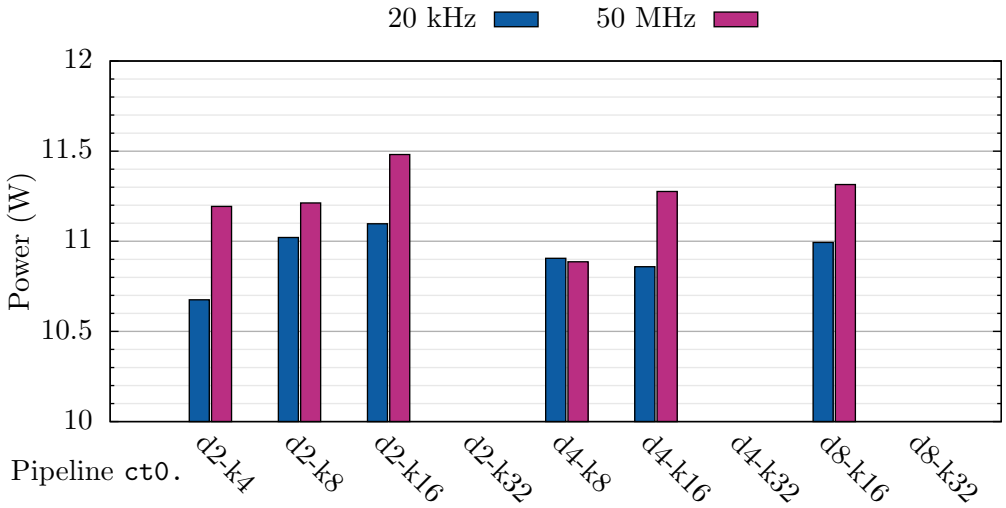



***Figure 5.17:*** *Pipeline architecture* `ct0.` *average power consumption of each Altera Stratix V GS 5SGSMD5 FPGA DSP development board implementation with pipeline enable* `en` *rate of 20 kHz and 50 MHz.*

Table 5.27 shows the average power consumption of each Altera Stratix V GS 5SGSMD5 FPGA DSP development board `ce0.` pipeline architecture implementation with pipeline enable `en` rate of 20 kHz and 50 MHz. Figure 5.18 illustrates the average power consumption of each Altera Stratix V GS 5SGSMD5 FPGA DSP development board `ce0.` pipeline architecture implementation with pipeline enable `en` rate of 20 kHz and 50 MHz.

***Table 5.27:*** *Pipeline architecture* `ce0.` *average power consumption of each Altera Stratix V GS 5SGSMD5 FPGA DSP development board implementation with pipeline enable* `en` *rate of 20 kHz and 50 MHz.*

| Pipeline `ce0.` | | | |
|---|---|---|---|
| $\bar{\boldsymbol{x}}$ Dimension ($d$) | Support Vectors ($k$) | Power (W) @ pipeline `en` rate: | |
| | | 20 kHz | 50 MHz |
| 2 | 4 | 10.80 | 10.85 |
| | 8 | 10.66 | 10.91 |
| | 16 | 11.05 | 10.87 |
| | 32 | 11.14 | 11.01 |
| 4 | 8 | 10.85 | 10.76 |
| | 16 | 10.91 | 11.30 |
| | 32 | 11.12 | 10.76 |
| 8 | 16 | 10.78 | 10.87 |
| | 32 | 10.87 | 10.72 |

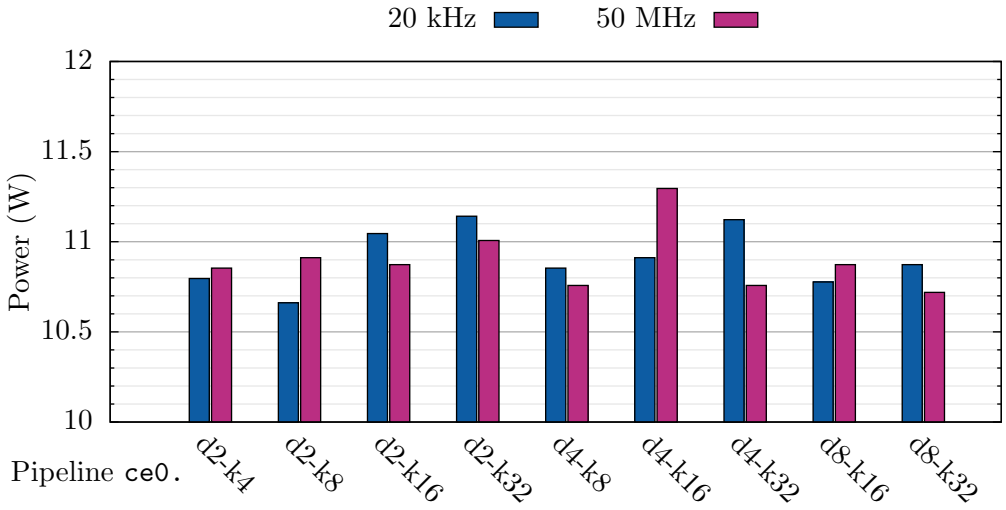



***Figure 5.18:*** *Pipeline architecture* `ce0.` *average power consumption of each Altera Stratix V GS 5SGSMD5 FPGA DSP development board implementation with pipeline enable* `en` *rate of 20 kHz and 50 MHz.*

Table 5.28 shows the average power consumption of each Altera Stratix V GS 5SGSMD5 FPGA DSP development board `rt0.` pipeline architecture implementation with pipeline enable `en` rate of 20 kHz and 50 MHz. Figure 5.19 illustrates the average power consumption of each Altera Stratix V GS 5SGSMD5 FPGA DSP development board `rt0.` pipeline architecture implementation with pipeline enable `en` rate of 20 kHz and 50 MHz.

***Table 5.28:*** *Pipeline architecture* `rt0.` *average power consumption of each Altera Stratix V GS 5SGSMD5 FPGA DSP development board implementation with pipeline enable* `en` *rate of 20 kHz and 50 MHz.*

| Pipeline `rt0.` | | | |
|---|---|---|---|
| $\bar{x}$ Dimension ($d$) | Support Vectors ($k$) | Power (W) @ pipeline `en` rate: | |
| | | 20 kHz | 50 MHz |
| 2 | 4 | 10.85 | 11.04 |
| | 8 | 10.89 | 10.98 |
| | 16 | 11.17 | 10.87 |
| | 32 | - | - |
| 4 | 8 | 11.08 | 11.04 |
| | 16 | 11.17 | 11.19 |
| | 32 | - | - |
| 8 | 16 | 10.98 | 11.19 |
| | 32 | - | - |

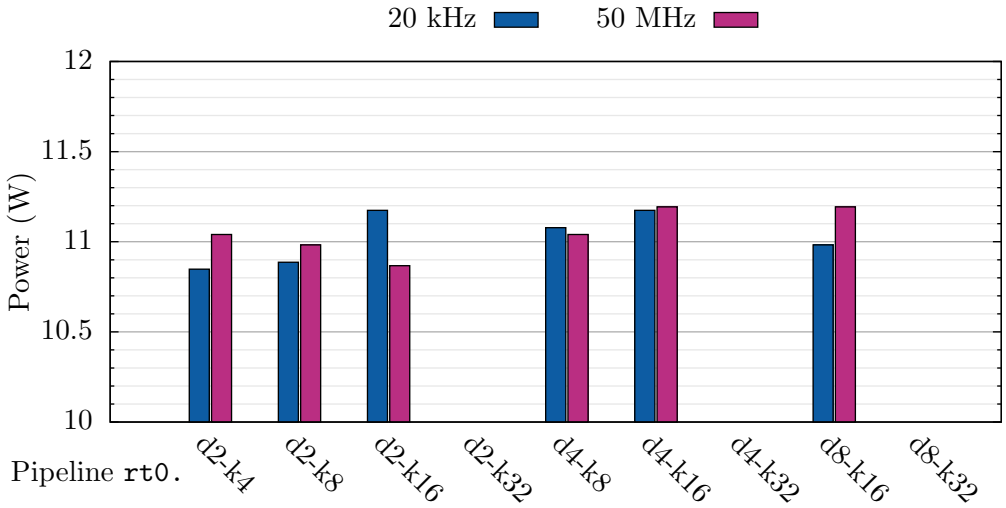



***Figure 5.19:*** *Pipeline architecture* `rt0.` *average power consumption of each Altera Stratix V GS 5SGSMD5 FPGA DSP development board implementation with pipeline enable* `en` *rate of 20 kHz and 50 MHz.*

Table 5.29 shows the average power consumption of each Altera Stratix V GS 5SGSMD5 FPGA DSP development board `re0.` pipeline architecture implementation with pipeline enable `en` rate of 20 kHz and 50 MHz. Figure 5.20 illustrates the average power consumption of each Altera Stratix V GS 5SGSMD5 FPGA DSP development board `re0.` pipeline architecture implementation with pipeline enable `en` rate of 20 kHz and 50 MHz.

***Table 5.29:*** *Pipeline architecture* `re0.` *average power consumption of each Altera Stratix V GS 5SGSMD5 FPGA DSP development board implementation with pipeline enable* `en` *rate of 20 kHz and 50 MHz.*

| Pipeline `re0.` | | | |
|---|---|---|---|
| $\bar{x}$ Dimension ($d$) | Support Vectors ($k$) | Power (W) @ pipeline `en` rate: | |
| | | 20 kHz | 50 MHz |
| 2 | 4 | 10.98 | 11.14 |
| | 8 | 11.06 | 11.14 |
| | 16 | 11.25 | 11.67 |
| | 32 | - | - |
| 4 | 8 | 11.06 | 11.50 |
| | 16 | 11.12 | 11.37 |
| | 32 | - | - |
| 8 | 16 | 11.08 | 11.33 |
| | 32 | - | - |

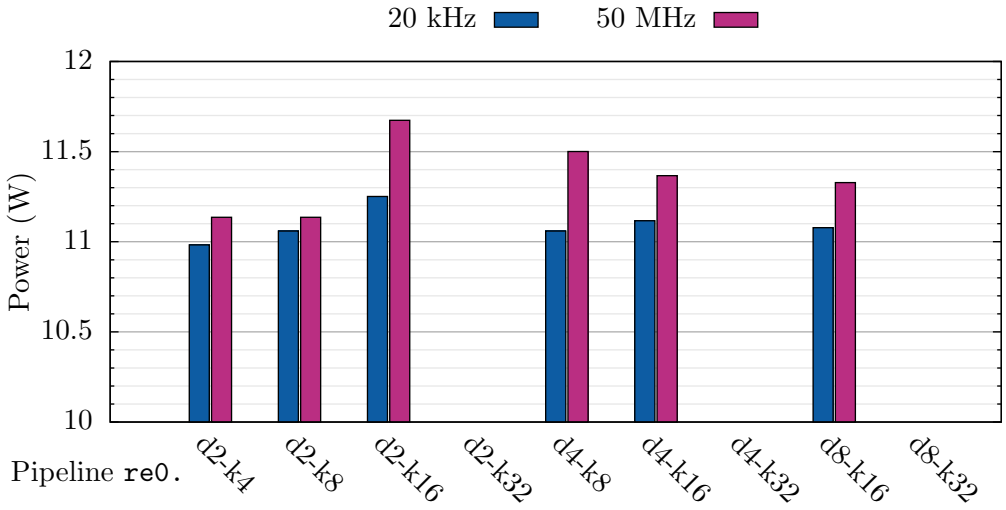



***Figure 5.20:*** *Pipeline architecture* `re0.` *average power consumption of each Altera Stratix V GS 5SGSMD5 FPGA DSP development board implementation with pipeline enable* `en` *rate of 20 kHz and 50 MHz.*

## 5.3 Machine Learning Results

The results in this section are arranged into subsections pertaining to the *Classification* and *Regression* machine learning experimental applications as shown in Table 4.11 and outlined in Section 4.4. Each subsection is further organised by data-set: *Legacy Pipeline Data* (LPD), *Lorenz Attractor Data* (LAD), *Mackey-Glass Attractor Data* (MGAD), and *Artificial Neural Network Data* (ANND).

### 5.3.1 SVM Classification Results

This section presents the optimal SVM classification results obtained through the application of the scientific methodologies with the four data-sets described in Section 4.4. The Linear kernel, Polynomial kernel, Radial Basis Function / Gaussian kernel, and Sigmoid kernel were applied with LIBSVM, however optimal results where obtained with the Polynomial kernel and thus only these results are reproduced here.

SVM classification results derived from the application of the brute-force training technique are absent from this section due to technological limitations when applied to non-trivial data-sets. Similarly combined exterior penalty function and barrier function optimisation method and the augmented Lagrange multiplier optimisation method is not shown due to unacceptably poor performance in both function and resulting classification accuracies. The failures of these proposed SVM classification training techniques will be discussed in Chapter 6.

SVM classification results using the DSP pipelines outlined in Section 4.3 are only shown when a LIBSVM-trained SVM classification model support vector set does not exceed the technical limitations of the `ce0.` DSP pipeline architecture variations.

#### 5.3.1.1 C-LPD

Optimal results obtained for SVM classification applied to the LPD data-set using LIBSVM training and function evaluation routines and utilising the polynomial kernel are shown in Table 5.30. Optimal results obtained for SVM classification applied to the LPD data-set using LIBSVM training and `ce0.` DSP pipeline function evaluation are shown in Table 5.31.

***Table 5.30:*** *C-LPD - SVM classification with LIBSVM training and function evaluation routines with the polynomial kernel; number of trained-model support vectors, training cross validation accuracy for $n = 100$ and training cost parameter $C = 1,000,000$, and test accuracy.*

| | $\bar{\boldsymbol{x}}$ Dimension = 2 | | | $\bar{\boldsymbol{x}}$ Dimension = 4 | | | $\bar{\boldsymbol{x}}$ Dimension = 8 | | |
|---|---|---|---|---|---|---|---|---|---|
| +1 Class | No. SVs | Train % | Test % | No. SVs | Train % | Test % | No. SVs | Train % | Test % |
| 0 | 98 | 95.52 | 96.27 | 18 | 99.46 | 99.51 | 9 | 99.77 | 99.92 |
| 1 | 138 | 93.39 | 90.20 | 89 | 98.06 | 97.53 | 69 | 99.38 | 99.42 |
| 2 | 6 | 100 | 99.84 | 16 | 100 | 99.92 | 9 | 99.77 | 99.60 |
| 3 | 6 | 99.85 | 99.84 | 5 | 100 | 99.92 | 6 | 100 | 99.92 |
| 4 | 4 | 100 | 99.92 | 4 | 100 | 100 | 9 | 100 | 99.92 |

***Table 5.31:*** *C-LPD - SVM classification with LIBSVM training and* `ce0`*. DSP pipeline function evaluation; number of trained-model support vectors, training cross validation accuracy for $n = 100$ and training cost parameter $C = 1000000$, and test accuracy. Cells shown in grey could not be computed due to the number of support vectors exceeding the pipeline hardware limitations.*

| | $\bar{\boldsymbol{x}}$ Dimension = 2 | | | $\bar{\boldsymbol{x}}$ Dimension = 4 | | | $\bar{\boldsymbol{x}}$ Dimension = 8 | | |
|---|---|---|---|---|---|---|---|---|---|
| +1 Class | No. SVs | Train % | Test % | No. SVs | Train % | Test % | No. SVs | Train % | Test % |
| 0 | 98 | - | - | 18 | - | - | 9 | 99.77 | 99.92 |
| 1 | 138 | - | - | 89 | - | - | 69 | - | - |
| 2 | 6 | 100 | 99.84 | 16 | 100 | 99.92 | 9 | 99.77 | 99.60 |
| 3 | 6 | 99.85 | 99.84 | 5 | 100 | 99.92 | 6 | 100 | 99.92 |
| 4 | 4 | 100 | 99.92 | 4 | 100 | 100 | 9 | 100 | 99.92 |

#### 5.3.1.2 C-LAD

Optimal results obtained for SVM classification applied to the LAD data-set using LIBSVM training and function evaluation routines and utilising the polynomial kernel are shown in Table 5.32.

***Table 5.32:*** *C-LAD - SVM classification with LIBSVM training and function evaluation routines with the polynomial kernel; number of trained-model support vectors, training cross validation accuracy for $n = 100$ and training cost parameter $C = 1,000,000$, and test accuracy.*

| | $\bar{\boldsymbol{x}}$ Dimension = 2 | | | $\bar{\boldsymbol{x}}$ Dimension = 4 | | | $\bar{\boldsymbol{x}}$ Dimension = 8 | | |
|---|---|---|---|---|---|---|---|---|---|
| +1 Class | No. SVs | Train % | Test % | No. SVs | Train % | Test % | No. SVs | Train % | Test % |
| 0 | 422 | 87.41 | 88.73 | 222 | 93.50 | 93.46 | 126 | 97.97 | 97.87 |
| 1 | 530 | 77.90 | 81.32 | 537 | 78.47 | 81.17 | 517 | 80.26 | 82.27 |
| 2 | 464 | 80.67 | 78.72 | 466 | 81.07 | 78.72 | 472 | 81.32 | 78.72 |
| 3 | 454 | 81.80 | 78.25 | 457 | 81.80 | 78.25 | 460 | 81.56 | 78.26 |
| 4 | 493 | 80.02 | 80.46 | 416 | 86.27 | 87.08 | 238 | 95.05 | 95.83 |

#### 5.3.1.3 C-MGAD

Optimal results obtained for SVM classification applied to the MGAD data-set using LIBSVM training and function evaluation routines and utilising the polynomial kernel are shown in Table 5.33.

***Table 5.33:*** *C-MGAD - SVM classification with LIBSVM training and function evaluation routines with the polynomial kernel; number of trained-model support vectors, training cross validation accuracy for $n = 100$ and training cost parameter $C = 1,000,000$, and test accuracy.*

| | $\bar{\boldsymbol{x}}$ Dimension = 2 | | | $\bar{\boldsymbol{x}}$ Dimension = 4 | | | $\bar{\boldsymbol{x}}$ Dimension = 8 | | |
|---|---|---|---|---|---|---|---|---|---|
| +1 Class | No. SVs | Train % | Test % | No. SVs | Train % | Test % | No. SVs | Train % | Test % |
| 0 | 519 | 78.64 | 80.64 | 522 | 79.36 | 80.64 | 518 | 80.48 | 82.15 |
| 1 | 533 | 78.80 | 80.80 | 527 | 79.20 | 80.80 | 534 | 79.20 | 80.80 |
| 2 | 516 | 78.72 | 79.92 | 511 | 80.16 | 79.92 | 507 | 80.08 | 79.92 |
| 3 | 511 | 78.55 | 80.08 | 513 | 79.92 | 80.08 | 510 | 79.44 | 79.92 |
| 4 | 468 | 80.48 | 78.57 | 473 | 81.45 | 78.57 | 480 | 81.53 | 78.41 |

#### 5.3.1.4 C-ANND

Optimal results obtained for SVM classification applied to the ANND data-set using LIBSVM training and function evaluation routines and utilising the polynomial kernel is shown in Table 5.34.

***Table 5.34:*** *C-ANND - SVM classification with LIBSVM training and function evaluation routines with the polynomial kernel; number of trained-model support vectors, training cross validation accuracy for $n = 100$ and training cost parameter $C = 1,000,000$, and test accuracy.*

| | $\bar{\boldsymbol{x}}$ Dimension = 2 | | | $\bar{\boldsymbol{x}}$ Dimension = 4 | | | $\bar{\boldsymbol{x}}$ Dimension = 8 | | |
|---|---|---|---|---|---|---|---|---|---|
| +1 Class | No. SVs | Train % | Test % | No. SVs | Train % | Test % | No. SVs | Train % | Test % |
| 0 | 205 | 92.44 | 93.23 | 191 | 94.21 | 94.59 | 192 | 94.21 | 94.43 |
| 1 | 329 | 89.39 | 89.41 | 44 | 98.47 | 98.73 | 35 | 98.88 | 98.57 |
| 2 | 426 | 83.84 | 84.32 | 373 | 85.69 | 86.31 | 320 | 88.67 | 88.22 |
| 3 | 199 | 93.41 | 94.75 | 136 | 95.10 | 95.62 | 72 | 97.67 | 97.21 |
| 4 | 404 | 84.00 | 84.71 | 383 | 84.81 | 84.71 | 387 | 84.33 | 85.27 |

### 5.3.2 SVM Regression Results

This section presents the optimal SVM regression results obtained through the application of the scientific methodologies with the four data-sets described in Section 4.4. The Linear kernel, Polynomial kernel, Radial Basis Function / Gaussian kernel, and Sigmoid kernel were applied with LIBSVM, however optimal results where obtained with the Polynomial kernel and thus only these results are reproduced here.

As with their SVM classification training counterparts SVM regression training efforts using the combined exterior penalty function and barrier function optimisation method

and the augmented Lagrange multiplier optimisation method are not shown due to unacceptably poor performance in both function and resulting classification accuracies. The failures of these proposed regression training techniques will be discussed in Chapter 6.

SVM regression results using the DSP pipelines outlined in Section 4.3 are only shown when a LIBSVM-trained SVM regression model support vector set does not exceed the technical limitations of the `re0.` DSP pipeline architecture variations.

#### 5.3.2.1 R-LPD

Optimal results obtained for SVM regression applied to the LPD data-set using LIBSVM training and function evaluation routines with the polynomial kernel in 2-dimensions for training cost parameter $C = 1,000$ and $\epsilon = 0.1$ are shown in Fig. 5.21; the number of support vectors = 831, the mean squared error = 0.09, and the squared correlation coefficient = 0.96.

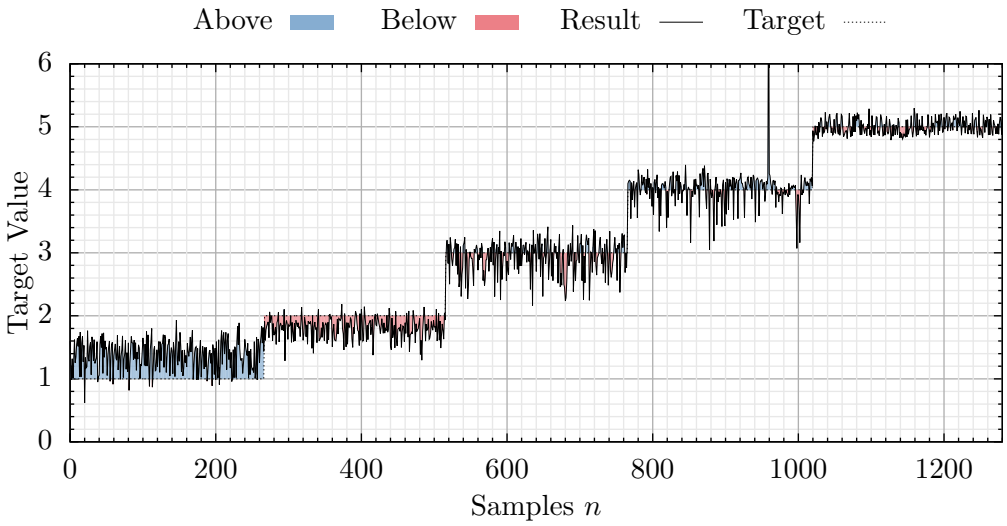



***Figure 5.21:*** *R-LPD - SVM regression with LIBSVM training and function evaluation routines with the polynomial kernel in 2-dimensions for training cost parameter* $C = 1,000$*, and* $\epsilon = 0.1$*; number of support vectors* $= 831$*, mean squared error* $= 0.09$*, and squared correlation coefficient* $= 0.96$*.*

Optimal results obtained for SVM regression applied to the LPD data-set using LIBSVM training and function evaluation routines with the polynomial kernel in 4-dimensions for training cost parameter $C = 108$ and $\epsilon = 0.09$ are shown in Fig. 5.22; the number of support vectors = 885, the mean squared error = 0.10, and the squared correlation coefficient = 0.95.

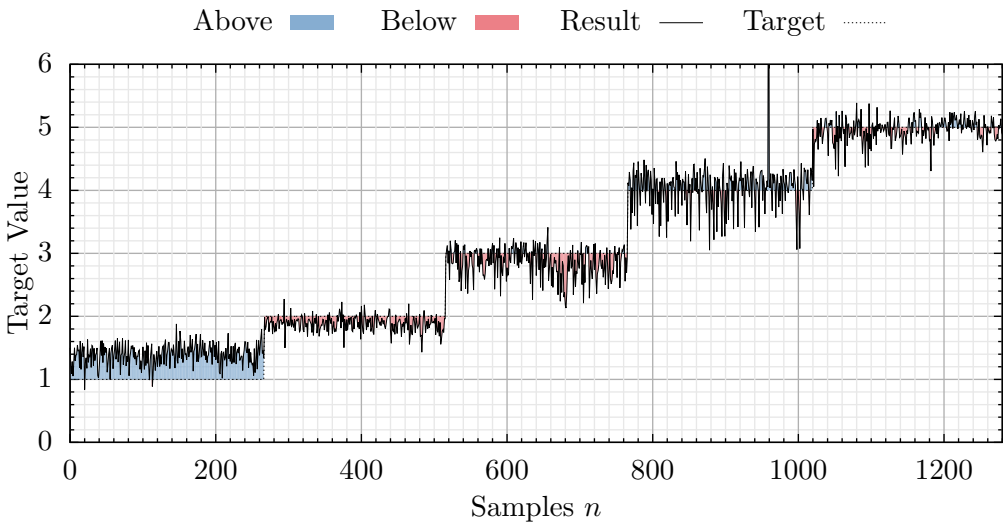



***Figure 5.22:*** *R-LPD - SVM regression with LIBSVM training and function evaluation routines with the polynomial kernel in 4-dimensions for training cost parameter* $C = 108$*, and* $\epsilon = 0.09$*; number of support vectors* $= 885$*, mean squared error* $= 0.10$*, and squared correlation coefficient* $= 0.95$*.*

Optimal results obtained for SVM regression applied to only two data-clusters of the LPD data-set using LIBSVM training and `re0`. DSP pipeline function evaluation in 4-dimensions for training cost parameter $C = 1000$ and $\epsilon = 0.1$ are shown in Fig. 5.23; the number of support vectors = 31, the mean squared error = 0.00, and the squared correlation coefficient = 0.99. Using only two data-clusters of the LPD data-set the trained model support vector count was kept within the technical limitations of of `re0`. DSP pipeline.

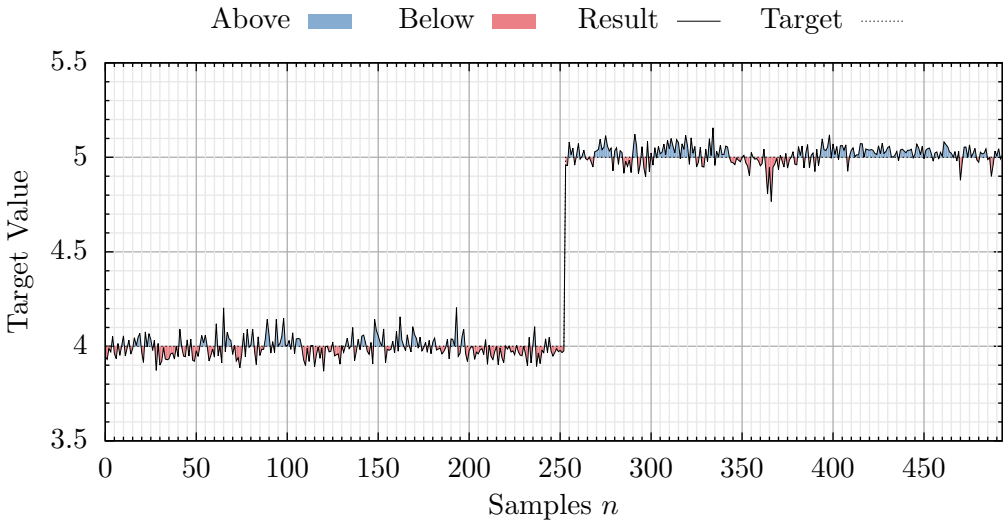



***Figure 5.23:*** *R-LPD - SVM regression with LIBSVM training (using to only two data-clusters of data-set to limit support vector count) and* `re0`*. DSP pipeline function evaluation in 4-dimensions for training cost parameter* $C = 1000$*, and* $\epsilon = 0.1$*; number of support vectors* $= 31$*, mean squared error* $= 0.00$*, and squared correlation coefficient* $= 0.99$*.*

Optimal results obtained for SVM regression applied to the LPD data-set using LIBSVM training and function evaluation routines with the polynomial kernel in 8-dimensions for training cost parameter $C = 6$, and $\epsilon = 0.1$ are shown in Fig. 5.24; the number of support vectors = 806, the mean squared error = 0.10, and the squared correlation coefficient = 0.96.

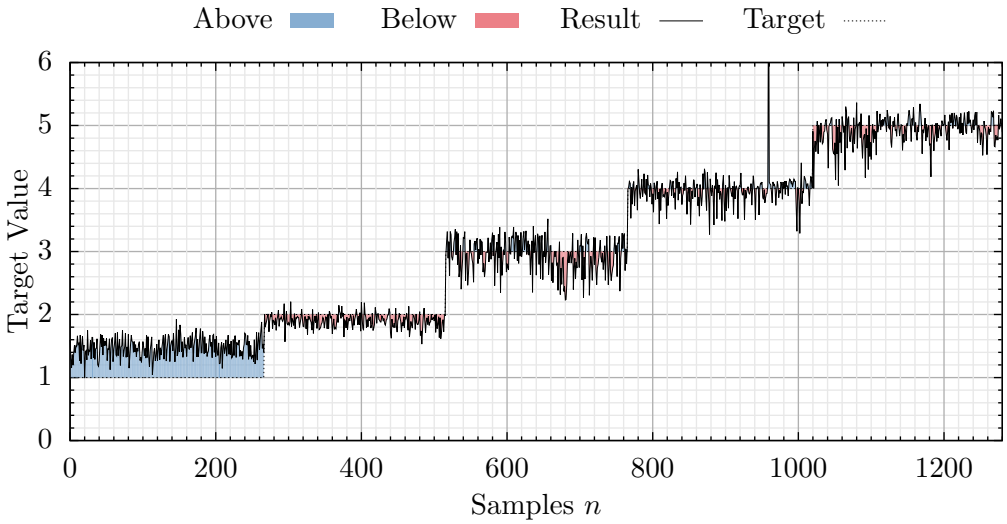



***Figure 5.24:*** *R-LPD - SVM regression with LIBSVM training and function evaluation routines with the polynomial kernel in 8-dimensions for training cost parameter $C = 6$, and $\epsilon = 0.1$; number of support vectors = 806, mean squared error = 0.10, and squared correlation coefficient = 0.96.*

Optimal results obtained for SVM regression applied to only two data-clusters of the LPD data-set using LIBSVM training and `re0.` DSP pipeline function evaluation in 8-dimensions for training cost parameter $C = 400$ and $\epsilon = 0.1$ are shown in Fig. 5.25; the number of support vectors = 30, the mean squared error = 0.00, and the squared correlation coefficient = 0.99. Using only two data-clusters of the LPD data-set the trained model support vector count was kept within the technical limitations of of `re0.` DSP pipeline.

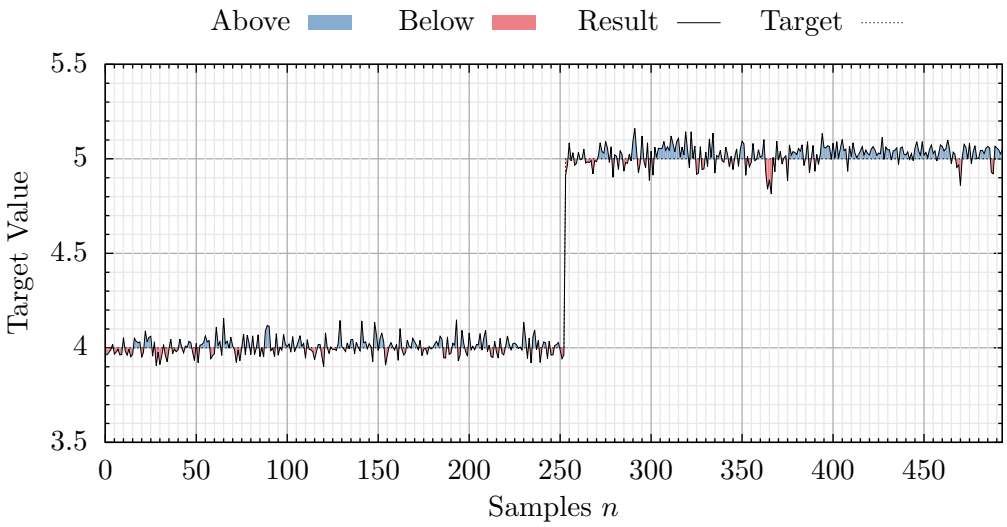



***Figure 5.25:*** *R-LPD - SVM regression with LIBSVM training (using to only two data-clusters of data-set to limit support vector count) and* `re0`*. DSP pipeline function evaluation in 8-dimensions for training cost parameter* $C = 400$*, and* $\epsilon = 0.1$*; number of support vectors* $= 30$*, mean squared error* $= 0.00$*, and squared correlation coefficient* $= 0.99$*.*

#### 5.3.2.2 R-LAD

Optimal results obtained for SVM regression applied to the LAD data-set using LIBSVM training and function evaluation routines with the polynomial kernel in 2-dimensions for training cost parameter $C = 1,000,000$, and $\epsilon = 0.1$ are shown in Fig. 5.26; the number of support vectors $= 1,229$, the mean squared error $= 7.07$, and the squared correlation coefficient $= 0.11$.

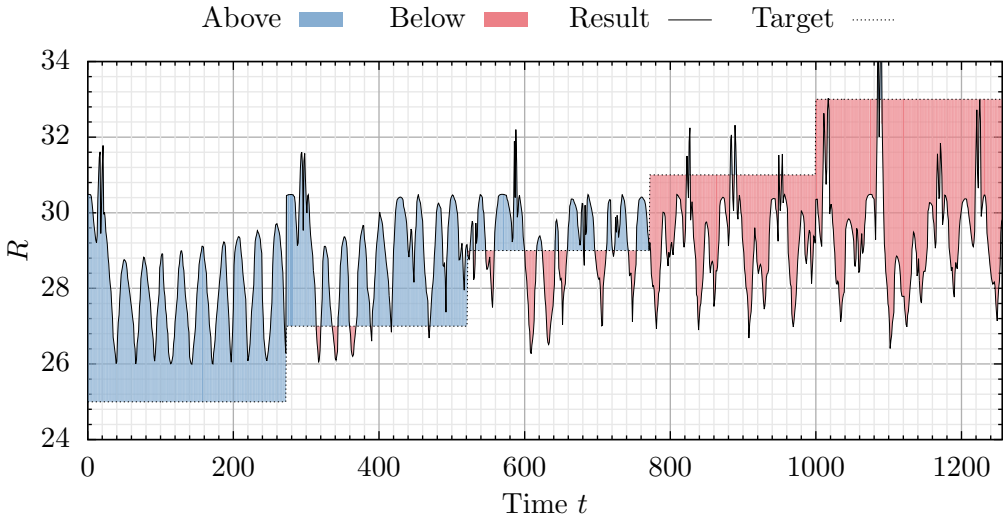



***Figure 5.26:*** *R-LAD - SVM regression with LIBSVM training and function evaluation routines with the polynomial kernel in 2-dimensions for training cost parameter* $C = 1,000,000$*, and* $\epsilon = 0.1$*; number of support vectors* $= 1,229$*, mean squared error* $= 7.07$*, and squared correlation coefficient* $= 0.11$*.*

Optimal results obtained for SVM regression applied to the LAD data-set using LIBSVM

training and function evaluation routines with the polynomial kernel in 4-dimensions for training cost parameter $C = 1,000,000$, and $\epsilon = 0.1$ are shown in Fig. 5.27; the number of support vectors $= 1,213$, the mean squared error $= 4.09$, and the squared correlation coefficient $= 0.51$.

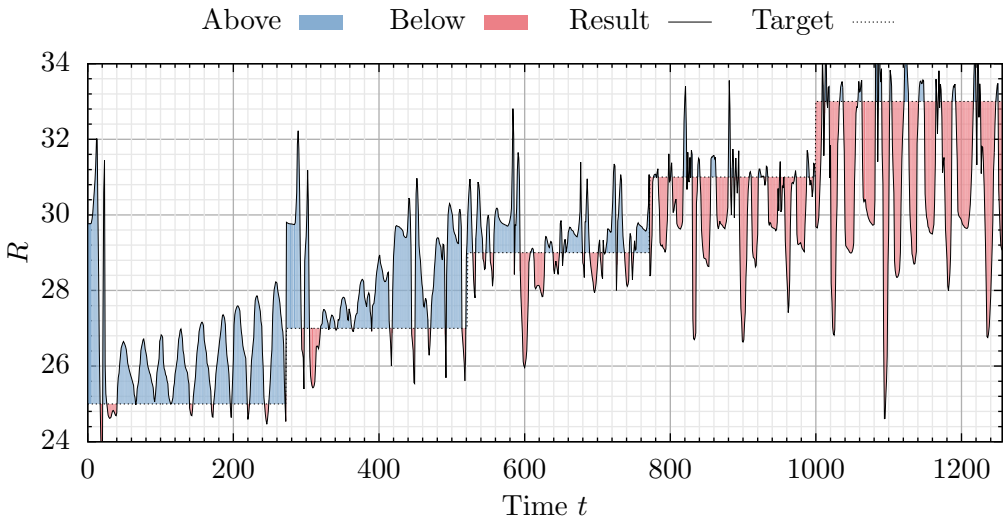



***Figure 5.27:*** *R-LAD - SVM regression with LIBSVM training and function evaluation routines with the polynomial kernel in 4-dimensions for training cost parameter* $C = 1,000,000$*, and* $\epsilon = 0.1$*; number of support vectors* $= 1,213$*, mean squared error* $= 4.09$*, and squared correlation coefficient* $= 0.51$*.*

Optimal results obtained for SVM regression applied to the LAD data-set using LIBSVM training and function evaluation routines with the polynomial kernel in 8-dimensions for training cost parameter $C = 1,000,000$, and $\epsilon = 0.1$ are shown in Fig. 5.28; the number of support vectors $= 1,223$, the mean squared error $= 2.09$, and the squared correlation coefficient $= 0.75$.

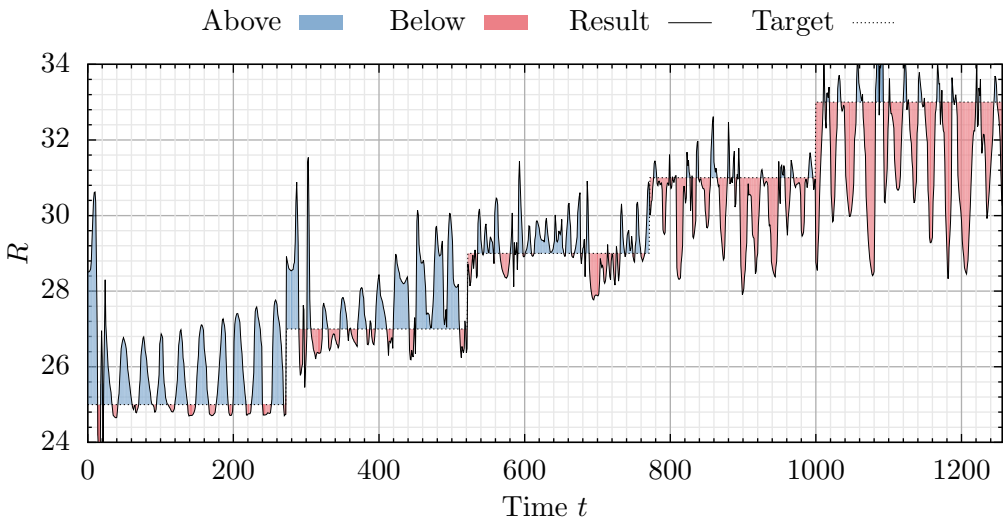



***Figure 5.28:*** *R-LAD - SVM regression with LIBSVM training and function evaluation routines with the polynomial kernel in 8-dimensions for training cost parameter* $C = 1,000,000$*, and* $\epsilon = 0.1$*; number of support vectors* $= 1,223$*, mean squared error* $= 2.09$*, and squared correlation coefficient* $= 0.75$*.*

#### 5.3.2.3 R-MGAD

Optimal results obtained for SVM regression applied to the MGAD data-set using LIBSVM training and function evaluation routines with the polynomial kernel in 2-dimensions for training cost parameter $C = 1,000,000$, and $\epsilon = 0.1$ are shown in Fig. 5.29; the number of support vectors $= 1,195$, the mean squared error $= 8.17$, and the squared correlation coefficient $= 0.01$.

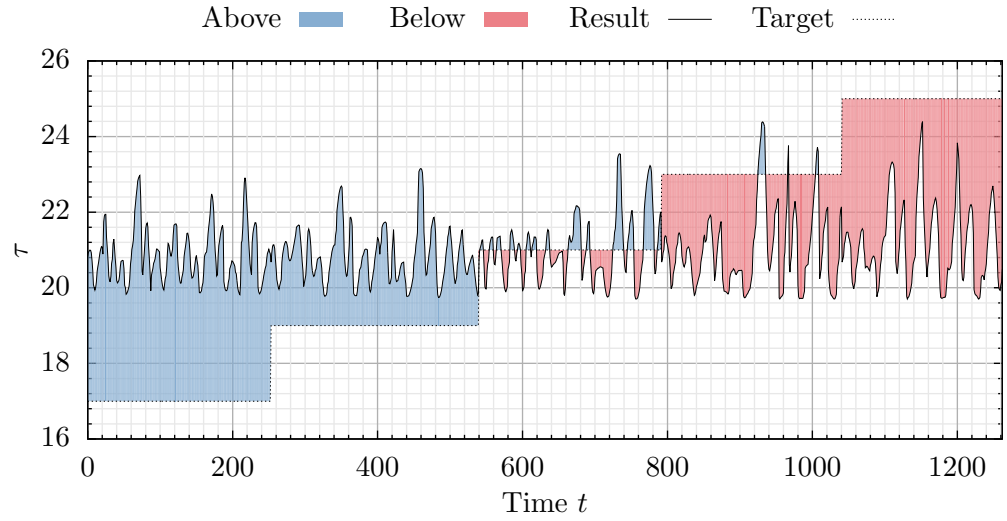



***Figure 5.29:*** *R-MGAD - SVM regression with LIBSVM training and function evaluation routines with the polynomial kernel in 2-dimensions for training cost parameter* $C = 1,000,000$*, and* $\epsilon = 0.1$*; number of support vectors* $= 1,195$*, mean squared error* $= 8.17$*, and squared correlation coefficient* $= 0.01$*.*

Optimal results obtained for SVM regression applied to the MGAD data-set using LIB-

SVM training and function evaluation routines with the polynomial kernel in 4-dimensions for training cost parameter $C = 1,000,000$, and $\epsilon = 0.1$ are shown in Fig. 5.30; the number of support vectors $= 1,187$, the mean squared error $= 7.62$, and the squared correlation coefficient $= 0.05$.

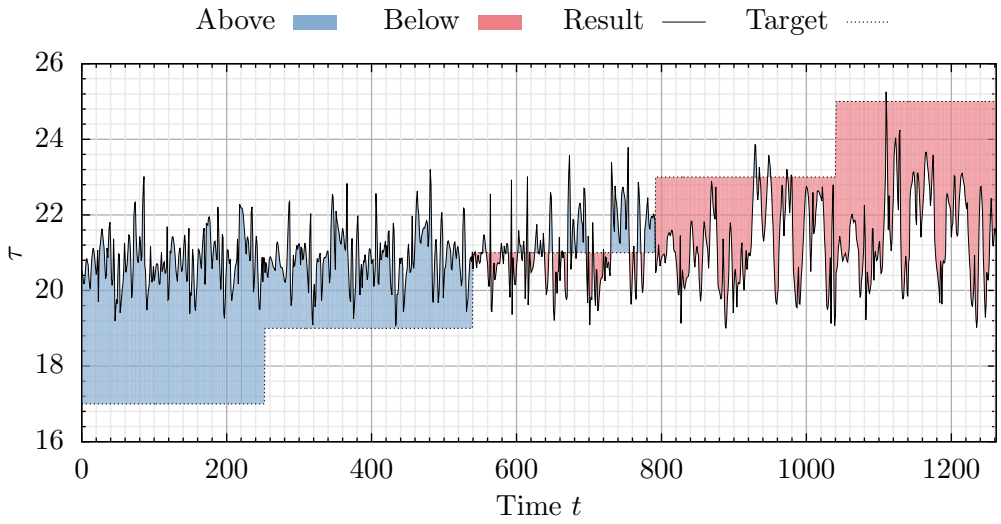



***Figure 5.30:*** *R-MGAD - SVM regression with LIBSVM training and function evaluation routines with the polynomial kernel in 4-dimensions for training cost parameter $C = 1,000,000$, and $\epsilon = 0.1$; number of support vectors $= 1,187$, mean squared error $= 7.62$, and squared correlation coefficient $= 0.05$.*

Optimal results obtained for SVM regression applied to the MGAD data-set using LIBSVM training and function evaluation routines with the polynomial kernel in 8-dimensions for training cost parameter $C = 1,000,000$, and $\epsilon = 0.1$ are shown in Fig. 5.31; the number of support vectors $= 1,187$, the mean squared error $= 6.67$, and the squared correlation coefficient $= 0.08$.

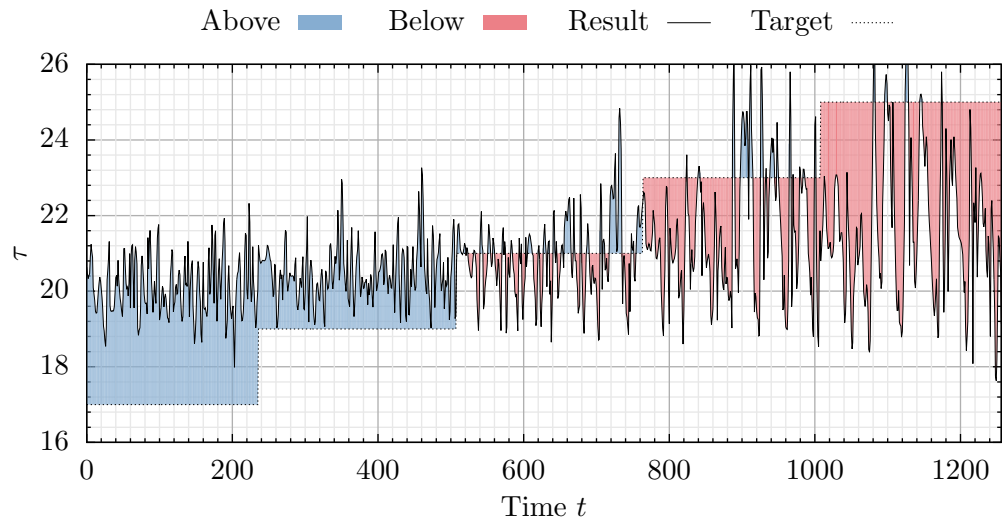



***Figure 5.31:*** *R-MGAD - SVM regression with LIBSVM training and function evaluation routines with the polynomial kernel in 8-dimensions for training cost parameter* $C = 1,000,000$*, and* $\epsilon = 0.1$*; number of support vectors* $= 1,222$*, mean squared error* $= 6.67$*, and squared correlation coefficient* $= 0.08$*.*

#### 5.3.2.4 R-ANND

Optimal results obtained for SVM regression applied to the ANND data-set using LIBSVM training and function evaluation routines with the polynomial kernel in 2-dimensions for training cost parameter $C = 1,000,000$, and $\epsilon = 0.1$ are shown in Fig. 5.32; the number of support vectors $= 1,243$, the mean squared error $= 8.17$, and the squared correlation coefficient $= 0.01$.

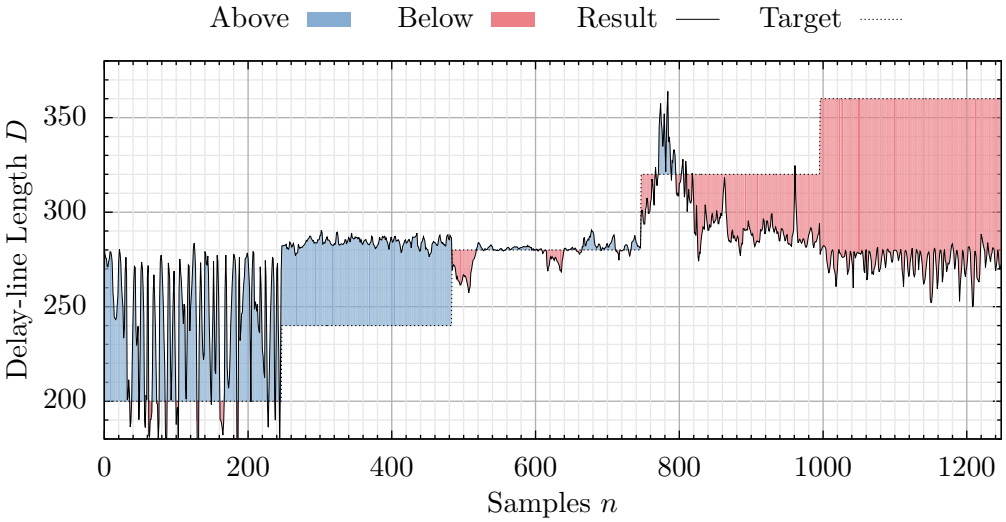



***Figure 5.32:*** *R-ANND - SVM regression with LIBSVM training and function evaluation routines with the polynomial kernel in 2-dimensions for training cost parameter* $C = 1,000,000$*, and* $\epsilon = 0.1$*; number of support vectors* $= 1,243$*, mean squared error* $= 8.17$*, and squared correlation coefficient* $= 0.01$*.*

Optimal results obtained for SVM regression applied to the ANND data-set using LIB-

SVM training and function evaluation routines with the polynomial kernel in 4-dimensions for training cost parameter $C = 1,000,000$, and $\epsilon = 0.1$ are shown in Fig. 5.33; the number of support vectors $= 1,248$, the mean squared error $= 7.62$, and the squared correlation coefficient $= 0.05$.

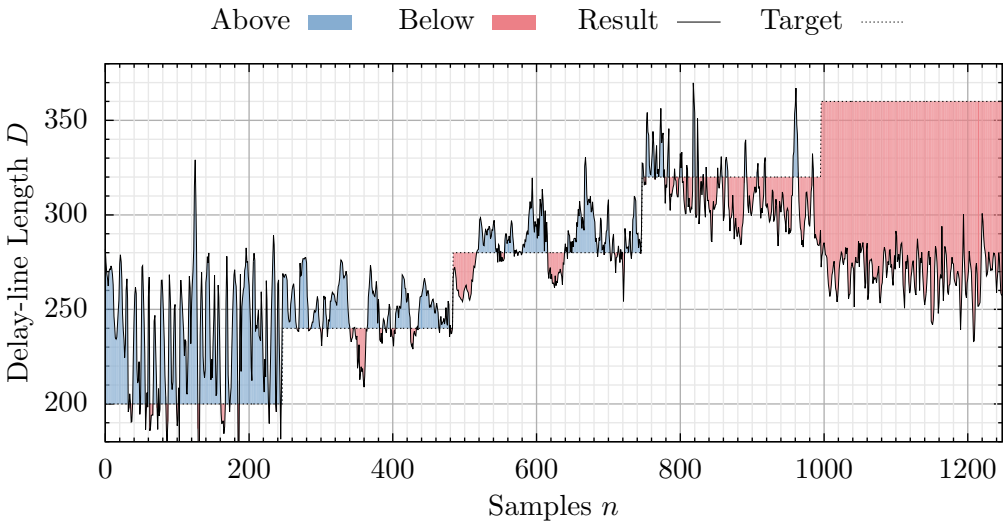



***Figure 5.33:*** *R-ANND - SVM regression with LIBSVM training and function evaluation routines with the polynomial kernel in 4-dimensions for training cost parameter $C = 1,000,000$, and $\epsilon = 0.1$; number of support vectors $= 1,248$, mean squared error $= 7.62$, and squared correlation coefficient $= 0.05$.*

Optimal results obtained for SVM regression applied to the ANND data-set using LIBSVM training and function evaluation routines with the polynomial kernel in 8-dimensions for training cost parameter $C = 1,000,000$, and $\epsilon = 0.1$ are shown in Fig. 5.34; the number of support vectors $= 1,247$, the mean squared error $= 6.67$, and the squared correlation coefficient $= 0.08$.

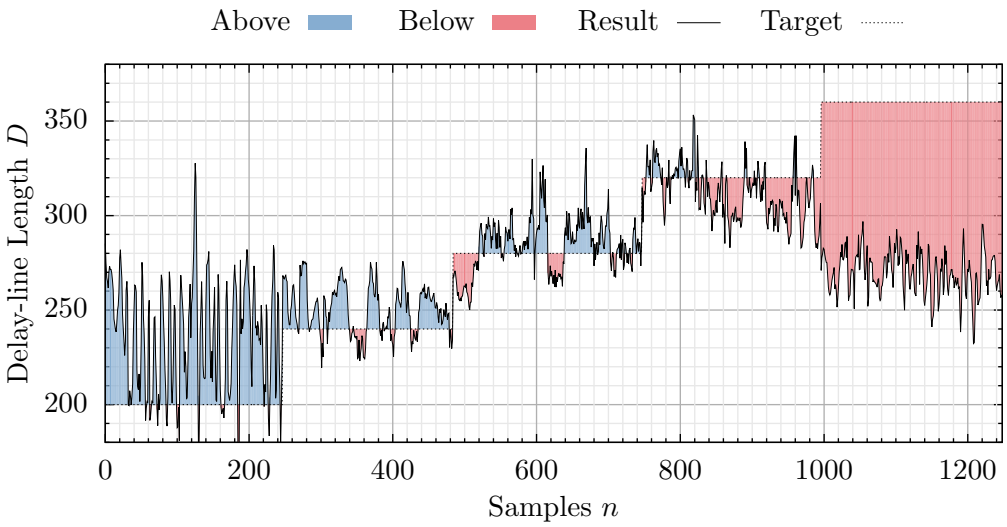



***Figure 5.34:*** *R-ANND - SVM regression with LIBSVM training and function evaluation routines with the polynomial kernel in 8-dimensions for training cost parameter* $C = 1,000,000$*, and* $\epsilon = 0.1$*; number of support vectors* $= 1,247$*, mean squared error* $= 6.67$*, and squared correlation coefficient* $= 0.08$*.*

# Chapter 6

# Discussion

This chapter provides an extensive discussion of the system designs and implementations presented in Chapter 4 and a discussion of the technical specifications, measured data, and results presented in the Chapter 5. The discussion in this chapter has been written considering this work's goals and objectives outlined in Chapter 1.

This chapter is organised into five sections; Section 6.1 Parallel-Architecture Training Discussion, Section 6.2 FPGA Hardware and DSP Pipeline Discussion, Section 6.3 DSP Results and Benchmarks Discussion, Section 6.4 Electrical Results Discussion, and finally Section 6.5 Machine Learning Results Discussion.

## 6.1 Parallel-Architecture Training Discussion

The three parallelised-architecture SVM training strategies developed as part of this work each exhibit fundamental failings in terms of their application in practical situations. This section will discuss these failings and why each architecture was not pursued further as part of this work.

The Brute-force search SVM training method becomes unwieldy to both implement and to instantiate or synthesize for problems requiring greater than 2-dimensional data-space with 4 support vectors. Compounding upon this as data-set size increases linearly the computation requirements increases exponentially.

Increasing the number of support vectors also posed a difficult problem - finding all potential support vector-combinations, as per Stage 2 of the design, for some arbitrary number of support vectors is not a trivial set of combination patterns to generate. It was found that the best solution, ironically, was applying a brute-force approach to find the complete set of support vector combination patterns. By letting each $\bar{\boldsymbol{x}}_i$ input vector in a training data-set of length $K$ be represented by 1 bit in a $K$-bit word, one can count from 0 to $2^{K-1}$ and test for Hamming Weight [105], the number of bits set in the word, of each increment higher. When the Hamming weight equals the number of support vectors a valid pattern has been found. All other bit-patterns are discarded. This job alone had

the potential to generate pattern data-sets gigabytes in size for training data-sets of only 500 samples in length.

One only needed to apply a problem of slightly larger data-set size than the trivial problems used to validate the brute-force search training designs as presented in Section 4.1.1 and the computational resource requirements became both unrealistic and certainly unimplementable on any target FPGA device platform. As discussed below in Section 6.2 the technological limitations and practicalities of current state-of-the-art FPGA technologies do not support any more than 5,760 DSP function blocks in the best case. This design was estimated to require upwards of 27,150 operations for the trivial cases alone. While this is not a great number of operations from a computational perspective - especially if one is considering implementing such a system on an High-performance Computing / Supercomputer (HPC) cluster - this design solution strategy was outside the scope of this work's goals and objectives, and well above the technological limitations of modern FPGA technology, thus, the brute-force search training solution was not perused any further.

The Combined Exterior Penalty and Interior Penalty / Barrier Function method and the Augmented Lagrange Multiplier mapped Neural Network optimisation techniques developed as part of this work do not suffer the same degree of computational complexity as the brute-force search training method. However both mapped-ANN training methods in their current form suffer their own flawed shortcomings. Both systems are inherently unstable. One is required to fine-tune several sets of gains for each parallelised neural pathway - if these gains are set too-high the system tends off to infinity after only a few iterations, if the gains are set too low the optimisation drifts indefinitely but never finds a feasible maximum. The author has considered implementing a PID or similar control system on each neural pathway and some hard bounding rules to both ensure a bounded system response and stable optimisation process. Whoever due to time constraints and adherence to the work's primary goals and objectives further work on these systems was adjourned.

This work has thus failed to solve the SVM training problem in a manner that provides real-time or even guaranteed fixed-period training latency on an FPGA device platform. However this work has provided a viable starting point for further work developing the parallelised architecture through ANN-mapped training strategies.

## 6.2 FPGA Hardware and DSP Pipeline Discussion

The FPGA DSP pipeline architectures designed and implemented in this work rely heavily on underlying proprietary FPGA fabric and embedded DSP function blocks to perform all arithmetic operations. These DSP function blocks are a finite resource on all of Altera's FPGA devices. As such the DSP pipeline architectures designed and implemented as part of this work were constrained by the volume of DSP block resources available on

the chosen Altera Stratix V FPGA device - the GS 5SGSMD5 FPGA. The Stratix V GS 5SGSMD5 has a total of 1,590 DSP clocks available in its FPGA fabric. Thus the finite DSP block resource count has a direct relationship on the number of parallel operations any given implementation can execute in the same clock-cycle. Due to this constraint the number of data-space dimensions and support vectors each DSP pipeline implementation can support is also limited. Table 5.3, Table 5.9, Table 5.15, and Table 5.21 provides a breakdown of each design-variation's FPGA resource requirements including DSP block resource utilisation. Compounding this design constraint further was the pragmatic reality of project funding and thus FPGA device availability at the project outset - at the time the Stratix V FPGA was the state-of-the-art device.

The following tables present all Altera FPGA devices supporting at-least the minimum dimension-support-vector SVM pipeline designs in terms of resource availability. The current state-of-the-art devices, the Altera Stratix 10 and Arria 10 FPGA devices and their relevant available resource totals are shown in Table 6.1, Table 6.2, Table 6.3, and Table 6.4. The previous generation Altera Stratix V (as used in this work), the mid-range Arria V, and entry-level Cyclone V FPGA devices and their key available resource totals are shown in Table 6.5, Table 6.6, Table 6.7, Table 6.8, and Table 6.9. It should be noted that the maximum available DSP blocks available in an Altera FPGA in the current market is 5,760 on the Startix 10 GX 2800 FPGA and SX 2800 system-on-a-chip. This is enough to accommodate the 8-dimension 32 support-vector designs the Stratix V GS 5SGSMD5 used in this work could not. It is however unlikely these devices could support the DSP block resource requirements for the step up to an 8-dimensional 64 support vector design.

***Table 6.1:*** *Altera Stratix 10 FPGA Resources.*

| Part Number | LEs | ALMs | ALM Registers | Variable-precision DSP Blocks |
|---|---|---|---|---|
| GX 500 | 484,000 | 164,160 | 656,640 | 1,152 |
| GX 650 | 646,000 | 218,880 | 875,520 | 1,440 |
| GX 850 | 841,000 | 284,960 | 1,139,840 | 2,016 |
| GX 1100 | 1,092,000 | 370,080 | 1,480,320 | 2,520 |
| GX 1650 | 1,624,000 | 550,540 | 2,202,160 | 3,145 |
| GX 2100 | 2,005,000 | 679,680 | 2,718,720 | 3,744 |
| GX 2500 | 2,422,000 | 821,150 | 3,284,600 | 5,011 |
| GX 2800 | 2,753,000 | 933,120 | 3,732,480 | 5,760 |
| GX 4500 | 4,463,000 | 1,512,820 | 6,051,280 | 1,980 |
| GX 5500 | 5,510,000 | 1,867,680 | 7,470,720 | 1,980 |

***Table 6.2:*** *Altera Stratix 10 SoC Resources.*

| Part Number | LEs | ALMs | ALM Registers | Variable-precision DSP Blocks |
|---|---|---|---|---|
| SX 500 | 484,000 | 164,160 | 656,640 | 1,152 |
| SX 650 | 646,000 | 218,880 | 875,520 | 1,440 |
| SX 850 | 841,000 | 284,960 | 1,139,840 | 2,016 |
| SX 1100 | 1,092,000 | 370,080 | 1,480,320 | 2,520 |
| SX 1650 | 1,624,000 | 550,540 | 2,202,160 | 3,145 |
| SX 2100 | 2,005,000 | 679,680 | 2,718,720 | 3,744 |
| SX 2500 | 2,422,000 | 821,150 | 3,284,600 | 5,011 |
| SX 2800 | 2,753,000 | 933,120 | 3,732,480 | 5,760 |
| SX 4500 | 4,463,000 | 1,512,820 | 6,051,280 | 1,980 |
| SX 5500 | 5,510,000 | 1,867,680 | 7,470,720 | 1,980 |

***Table 6.3:*** *Altera Arria 10 FPGA Resources.*

| Part Number | LEs | ALMs | Registers | Floating-point DSP Blocks |
|---|---|---|---|---|
| GX 160 | 160,000 | 61,510 | 246,040 | 156 |
| GX 220 | 220,000 | 83,730 | 334,920 | 191 |
| GX 270 | 270,000 | 101,620 | 406,480 | 830 |
| GX 320 | 320,000 | 118,730 | 474,920 | 985 |
| GX 480 | 480,000 | 181,790 | 727,160 | 1,368 |
| GX 570 | 570,000 | 217,080 | 868,320 | 1,523 |
| GX 660 | 660,000 | 250,540 | 1,002,160 | 1,688 |
| GX 900 | 900,000 | 339,620 | 1,358,480 | 1,518 |
| GX 1150 | 1,150,000 | 427,200 | 1,708,800 | 1,518 |
| GT 900 | 900,000 | 339,620 | 1,358,480 | 1,518 |
| GT 1150 | 1,150,000 | 427,200 | 1,708,800 | 1,518 |

***Table 6.4:*** *Altera Arria 10 SoC Resources.*

| Part Number | LEs | ALMs | Registers | Floating-point DSP Blocks |
|---|---|---|---|---|
| SX 160 | 160,000 | 61,510 | 246,040 | 156 |
| SX 220 | 220,000 | 83,730 | 334,920 | 191 |
| SX 270 | 270,000 | 101,620 | 406,480 | 830 |
| SX 320 | 320,000 | 118,730 | 474,920 | 985 |
| SX 480 | 480,000 | 181,790 | 727,160 | 1,368 |
| SX 570 | 570,000 | 217,080 | 868,320 | 1,523 |
| SX 660 | 660,000 | 250,540 | 1,002,160 | 1,688 |

***Table 6.5:*** *Altera Stratix V FPGA Resources.*

| Part Number | LEs | ALMs | Registers | Variable-precision DSP Blocks |
|---|---|---|---|---|
| 5SGSD3 | 236,000 | 89,000 | 356,000 | 600 |
| 5SGSD4 | 360,000 | 135,840 | 543,360 | 1,044 |
| 5SGSD5 | 457,000 | 172,600 | 690,400 | 1,590 |
| 5SGSD6 | 583,000 | 220,000 | 880,000 | 1,775 |
| 5SGSD8 | 695,000 | 262,400 | 1,049,600 | 1,963 |
| 5SGXA3 | 340,000 | 128,300 | 513,200 | 256 |
| 5SGXA4 | 420,000 | 158,500 | 634,000 | 256 |
| 5SGXA5 | 490,000 | 185,000 | 740,000 | 256 |
| 5SGXA7 | 622,000 | 234,720 | 938,880 | 256 |
| 5SGXA9 | 840,000 | 317,000 | 1,268,000 | 352 |
| 5SGXAB | 952,000 | 359,200 | 1,436,800 | 352 |
| 5SGXB5 | 490,000 | 185,000 | 740,000 | 399 |
| 5SGXB6 | 597,000 | 225,400 | 901,600 | 399 |
| 5SGXB9 | 840,000 | 317,000 | 1,268,000 | 352 |
| 5SGXBB | 952,000 | 359,200 | 1,436,800 | 352 |
| 5SGTC5 | 425,000 | 160,400 | 641,600 | 256 |
| 5SGTC7 | 622,000 | 234,720 | 938,880 | 256 |
| 5SEE9 | 840,000 | 317,000 | 1,268,000 | 352 |
| 5SEEB | 952,000 | 359,200 | 1,436,800 | 352 |

***Table 6.6:*** *Altera Arria V FPGA Resources.*

| Part Number | LEs | ALMs | Registers | Variable-precision DSP Blocks |
|---|---|---|---|---|
| 5AGXA1 | 75 | 28,302 | 113,208 | 240 |
| 5AGXA3 | 156 | 58,900 | 235,600 | 396 |
| 5AGXA5 | 190 | 71,698 | 286,792 | 600 |
| 5AGXA7 | 242 | 91,680 | 366,720 | 800 |
| 5AGXB1 | 300 | 113,208 | 452,832 | 920 |
| 5AGXB3 | 362 | 136,880 | 547,520 | 1,045 |
| 5AGXB5 | 420 | 158,491 | 633,964 | 1,092 |
| 5AGXB7 | 504 | 190,240 | 760,960 | 1,156 |
| 5AGTC3 | 156 | 58,900 | 235,600 | 396 |
| 5AGTC7 | 242 | 91,680 | 366,720 | 800 |
| 5AGTD3 | 362 | 136,880 | 547,520 | 1,045 |
| 5AGTD7 | 504 | 190,240 | 760,960 | 1,156 |
| 5AGZE1 | 220 | 83,020 | 332,080 | 800 |
| 5AGZE3 | 360 | 135,840 | 543,360 | 1,044 |
| 5AGZE5 | 400 | 150,960 | 603,840 | 1,092 |
| 5AGZE7 | 450 | 169,800 | 679,200 | 1,139 |

***Table 6.7:*** *Altera Arria V SoC Resources.*

| Part Number | LEs | ALMs | Registers | Variable-precision DSP Blocks |
|---|---|---|---|---|
| 5ASXB3 | 350 | 132,075 | 528,300 | 809 |
| 5ASXB5 | 462 | 174,340 | 697,360 | 1,090 |
| 5ASTD3 | 350 | 132,075 | 528,300 | 809 |
| 5ASTD5 | 462 | 174,340 | 697,360 | 1,090 |

***Table 6.8:*** *Altera Cyclone V FPGA Resources.*

| Part Number | LEs | ALMs | Registers | Variable-precision DSP Blocks |
|---|---|---|---|---|
| 5CEA4 | 49,000 | 18,480 | 73,920 | 66 |
| 5CEA5 | 77,000 | 29,080 | 116,320 | 150 |
| 5CEA7 | 149,500 | 56,480 | 225,920 | 156 |
| 5CEA9 | 301,000 | 113,560 | 454,240 | 342 |
| 5CGXC3 | 35,500 | 13,460 | 53,840 | 57 |
| 5CGXC4 | 50,000 | 18,868 | 75,472 | 70 |
| 5CGXC5 | 77,000 | 29,080 | 116,320 | 150 |
| 5CGXC7 | 149,500 | 56,480 | 225,920 | 156 |
| 5CGXC9 | 301,000 | 113,560 | 454,240 | 342 |
| 5CGTD5 | 77,000 | 29,080 | 116,320 | 150 |
| 5CGTD7 | 149,500 | 56,480 | 225,920 | 156 |
| 5CGTD9 | 301,000 | 113,560 | 454,240 | 342 |

***Table 6.9:*** *Altera Cyclone V SoC Resources.*

| Part Number | LEs | ALMs | Registers | Variable-precision DSP Blocks |
|---|---|---|---|---|
| 5CSEA4 | 40,000 | 15,094 | 60,376 | 84 |
| 5CSEA5 | 85,000 | 32,075 | 128,300 | 87 |
| 5CSEA6 | 110,000 | 41,509 | 166,036 | 112 |
| 5CSXC4 | 40,000 | 15,094 | 60,376 | 84 |
| 5CSXC5 | 85,000 | 32,075 | 128,300 | 87 |
| 5CSXC6 | 110,000 | 41,509 | 166,036 | 112 |
| 5CSTD5 | 85,000 | 32,075 | 128,300 | 87 |
| 5CSTD6 | 110,000 | 41,509 | 166,036 | 112 |

The limitation set on the maximum number of model support vectors exposed in this work is a reflection of current FPGA technological capacity and limitations. Based upon these state-of-the-art technological resource constraints posed by currently available FPGA technology this work has thus achieved a reasonable SVM design standard and made good use, in the name of minimising pipeline latency and achieving real-time performance, of the technology available at the time of this work.

It should be noted that for applications requiring less data-space dimensionality and feature-space complexity than the applications found in this work, and with tight bud-

getary restrictions, the low-end Cyclone V series FPGA can be utilised and will comfortably accommodate the resource requirements of this work's lower-dimensional DSP pipeline designs.

## 6.3 DSP Results and Benchmarks Discussion

The primary objective of this work was to implement a series of SVM architectures capable of operating in real-time; minimise each SVM construct's execution time or latency - the time between data entering the pipeline and a result appearing on the output. Table 5.2, Table 5.8, Table 5.14, and Table 5.20, respectively, presents each of the different SVM DSP pipeline architecture variation's pipeline length, measured in stages, and, the pipeline latency. Observing Table 5.4 and Fig. 5.1, Table 5.10 and Fig. 5.5, Table 5.16 and Fig. 5.9, and Table 5.10 and Fig. 5.5, respectively, one can see that the FPGA hardware DSP pipeline implementation latency is almost three orders of magnitude smaller than that of the four microprocessor architecture software model counterpart's execution time. The FPGA hardware DSP implementations have certainly minimised each pipeline's latency or execution time substantially.

It is interesting to note the variation in execution time standard-deviation of the software models across the various microprocessor platforms. The DSP pipelines implemented as FPGA hardware do not suffer from such execution delay variations - the pipeline latency is as constant as the FPGA system clock signal. This reinforces the FPGA implementations' suitability for deployment in not only industrial process applications but in time critical environments where the real-time consistent meeting of deadlines is of utmost importance, and, can have a bearing on both human operator and asset safety.

In the worst case the FPGA hardware DSP pipeline implementation latency is barely more than the software model's execution time standard-deviation by only fractions of a microsecond. The FPGA hardware DSP pipelines outclass the single-threaded software model counterparts across all microprocessor architecture platforms.

Observing the Instructions-per-Cycle of each FPGA hardware DSP pipeline implementation and their corresponding software model counterparts in Table 5.6 and Fig. 5.3, Table 5.12 and Fig. 5.7, Table 5.18 and Fig. 5.11, and Table 5.24 and Fig. 5.15, respectively, it is obvious to see where the performance gains in latency / execution time have come from - the number of operations being executed in parallel on the FPGA hardware is anywhere from over two to over four orders of magnitude greater than that of the software model.

Also, noting Table 5.1 CPU Cores column, each microprocessor architecture is, at best case, and, assuming a perfect multi-threaded DSP pipeline software model, capable of only 8 instructions-per-cycle for the Intel Core i7 6700K platform, 2 instructions-per-cycle for the Intel Core 2 Duo P8600 platform, 4 instructions-per-cycle for the Intel Atom

N570 platform, and 4 instructions-per-cycle for the Broadcom BCM2836 - ARM Cortex-A7 platform, respectively. These best case conditions are, however, not achievable due to operating system processing and multi-threading library message-passing overheads. Thus the software models running on the four microprocessor architectures cannot even come close to achieving a comparable Instructions-per-Cycle metric as that of the FPGA hardware DSP pipelines.

Table 5.3, Table 5.9, Table 5.15, and Table 5.21 provides a breakdown of each design-variation's FPGA resource requirements. Apart from using an increasingly significant portion of the Altera Stratix V GS 5SGSMD5 FPGA device's embedded DSP block resources as device variation parameters increase, the DSP pipeline architectures do not come close to using even half of the FPGA device's available hardware resources. This may be useful in future architecture design extensions for increasing the number of support vectors, at the expense of latency, by using the unused programmable digital logic FPGA fabric to effectively resource-share the valuable and scarce DSP functional blocks.

## 6.4 Electrical Results Discussion

Observing Table 5.26 and Fig. 5.17, Table 5.27 and Fig. 5.18, Table 5.28 and Fig. 5.19, and Table 5.29 and Fig. 5.20, respectively, FPGA device DSP pipeline power consumption always falls between 10.5 Watts and 11.5 Watts. As FPGA device resource usage increases so too does the power consumption. Also as pipeline enable `en` rate increases from 20 kHz to 50 MHz so too does the power consumption. However this is not always the case. This could be due to how Altera Quartus Prime development suite has optimised and synthesised each respective SVM circuit, and, then fitted the optimised circuit binary image onto the FPGA. It is feasible that certain areas of the FPGA fabric consume power at different rates - different IO pins on the FPGA package do operate at different voltage standards. Thus the anomaly cases where the 20 kHz pipeline enable `en` rate may be due to obscure FPGA implementation quirks or circuit optimisation and hardware fitting peculiarities.

The power consumption results reinforce the the case for the FPGA implementations' potential and suitability for deployment in industrial process applications and battery or combination solar-and-battery installations out in the field.

## 6.5 Machine Learning Results Discussion

Classification of the Legacy Pipeline LPD data-set using both LIBSVM and the `ce0.` SVM DSP pipeline, as presented in Table 5.30 and Table 5.31, show very good classification accuracy. The underlying mathematics of both LIBSVM and `ce0.` DSP pipeline are exactly the same, thus when presented with the same set of support vectors and test-data the results are identical. Due to the increased complexity in data-space where class 0 and

class 1 intersect, as can be observed by inspecting the cyan and dark-purple data-clusters in Fig. 4.35, the number of support vectors required to define an accurate margin between the two classes in both 2 and 4 dimensional data-space, and for class 1 in 8 dimensional data-space, exceeds the maximum support vector constraints for the `ce0.` SVM DSP pipeline - thus classification has not been conducted in these cases.

Table 5.32, and Table 5.34 show the classification results of the Lorenz Attractor LAD chaotic system data-set and the ANN Chaotic Oscillator ANND data-set, respectively. Generally as data-space dimensionality increases, both the LAD data-set classification accuracy and the ANND data-set accuracy increases - both data sets show a reasonable improvement in classification accuracy, from mediocre accuracy to good accuracy, as the the dimensionality increases. The 100 dimension state-space embedding and subsequent kPCA dimensionality reduction has done a reasonable job of embedding the very subtle changes in state-space evolution due to parameter variations into each data-point - enough to distinguish the majority of one class from the other.

However, data-space complexity was very high as reflected by the high number of support vectors required to define the margin between each class. Due to this high complexity and subsequent support vector volume the `ce0.` DSP pipeline could not be applied to these problems. A small drop in support vectors is observed as dimensionality is increased, however, indicating that the higher the dimension of data-space the more separation in space exists between each respective class - which one would expect considering the geometric and spacial implications of a higher dimensionality data-space.

The classification accuracies of the Mackey-Glass Attractor MGAD data-set, as shown in Table 5.33, does not improve as data-space dimensionality is increased. Also the data-space complexity does not decrease as the dimensionality increases as indicated by the relatively stable number of support vectors across the three data-space dimensions. Again due to the high complexity and subsequent support vector volume the `ce0.` DSP pipeline could not be applied to this problem. The 100 dimension state-space embedding and subsequent kPCA dimensionality reduction has achieved very little in the way of embedding the very subtle changes in state-space evolution due to parameter variations into each data-point - this may be due to the fact that the Mackey-Glass chaotic system is itself defined by a more complex state-space embedding as shown in Eq. 3.122.

Regression of the Legacy Pipeline data-set LPD using LIBSVM, as shown in Fig. 5.21, Fig. 5.22, and Fig. 5.24 reveal good target function tracking, a small mean squared error for each data-space dimensionality regression, and an excellent squared correlation coefficient. The blips in the result trace are due to outlier data-points in the test-data set. However once again a very high data-space complexity is revealed through the high volume of support vectors.

To work around the high support vector volume and adequately apply the `re0.` DSP pipeline the training data-set was simplified by reducing the number of regression target data-points. The reduced-complexity 2 dimensional data-space was still too complex to meet the low support vector requirements of the `re0.` DSP pipeline. Figure 5.23 and Fig. 5.25 display very good tracking of the reduced-complexity target regression function for the 4 dimension data-space and the 8 dimension data-space respectively. There is a very small mean squared error and an excellent squared correlation coefficient for both regressions.

Again, much like the classification of the chaotic data-sets the data-space complexity was much too high as reflected by the high number of support vectors required to define the regression models. Due to this high complexity and subsequent support vector volume the `re0.` DSP pipeline could not be applied to the chaotic system regression problems. However unlike the chaotic system's classification models there is no significant pattern in model support vector volume as dimensionality was increased.

The LAD data-set regressions shown in Fig. 5.26, Fig. 5.27, and Fig. 5.28 show that as data-set dimensionality increases the tracking of the target regression improves from very poor tracking in the 2 dimensional data-space to approaching an almost mediocre tracking in the 8 dimensional data-space. The mean squared error metric and squared correlation coefficient metric also improve as dimensionality is increased. The regression does not however provide an accurate identification of the $R$ parameter due to the high oscillations around the target function.

The ANND data-set regressions shown in Fig. 5.32, Fig. 5.33, and Fig. 5.34 show that as data-set dimensionality increases the tracking of the target regression improves but only for mid-range values of $D$ from very poor tracking in the 2 dimensional data-space to very poor oscillating tracking in the 8 dimensional data space. The mean squared error metric and squared correlation coefficient metric do not improve as dimensionality is increased. The regression does not provide an accurate identification of the $D$ parameter due to the high errors or high oscillations around the target function.

The MGAD data-set regressions shown in Fig. 5.29, Fig. 5.30, and Fig. 5.31 show almost no tracking of the target function other than a noisy regression about the mean of the target. As data-set dimensionality increases the noise about the target mean becomes noisier. The mean squared error metric and squared correlation coefficient metric do not improve as dimensionality is increased. The regression does not provide an accurate identification of the $\tau$ parameter due to the completely unsuccessful tracking of the target function.

SVM regression has performed very poorly across the three chaotic data-sets. SVM classification of the preprocessed LAD and ANND data-sets returned adequate results.

The `ce0.` and `re0.` DSP pipelines, while being constrained to low-complexity data-space problems due to their low support vector volume limitations performed the tasks they where capable of fulfilling exceptionally well.

# Chapter 7

# Conclusion

## 7.1 Recommendations and Future Work

This section provides a brief presentation of potential future work and research efforts that have become apparent through the progression of this research project, have not yet been explored due to limited project time constraints, fall outside the scope of this work's immediate goals and objectives, or serves as a compliment or logical next step along the same line of research exploration this project has followed.

The SVM DSP pipeline implementations realised as part of this body of work can be further extended. The polynomial kernel can be uncoupled from the SVM DSP pipeline designs and thus form separate modular polynomial kernel pipelines and SVM DSP pipelines. While uncoupling the polynomial kernel from the SVM pipelines in this manner will in some SVM pipeline cases lead to greater pipeline latency due to diminished pipelined-stage optimisation, alternative kernel pipelines, as presented in Chapter 4, can also be employed as direct modular replacements to the polynomial kernel in the cascaded kernel-SVM pipeline architecture.

Additionally at the potential expense of pipeline latency and increased pipeline design complexity and design timing requirements the SVM pipeline designs can be extended to accommodate higher dimensional data-space data-sets and increased model support vector compatibility.

The DSP pipeline software models can be modified to incorporate multi-threaded or parallelised implementation elements utilising OpenCL (CPU and GPGPU architectures), Nvidia CUDA (GPGPU), and OpenMP and MPI (multi-core CPU, HPC, and commodity-hardware clusters) libraries. Then low power ARM-based commodity hardware, such as the Raspberry Pi 3 single-board computer, can be utilised in large volume to create relatively cheap and low-power HPC clusters for the parallelised SVM software model architectures to run on. While a system of this kind increases hardware overhead and system components this solution may prove to be cheaper than that of high-end FPGAs with the required abundance of DSP function-blocks, allow the use of 64-bit double-

precision floating-point operations, and, provide the all the benefits of the system being implemented wholly in software - thus is easier to extend, maintain, and debug than the VHDL-described hardware counterparts implemented in this work.

Both $k$-means clustering and PCA / kPCA can also be implemented as DSP pipelines and thus realised as novel FPGA DSP pipeline hardware.

Finally the hardware SVM test-rig can extended to include additional high-speed communications systems such as $I^2C$, SPI, and PCI Express for high-speed communication with a host PC for improved DSP pipeline utilisation as an auxiliary computational engine and result reporting, and for use in industrial process control PLC architectures.

## 7.2 Conclusions

The objective of this research project was to investigate, implement, and apply SVM classification and regression machine learning paradigms capable of operating in real-time. This objective was met by implementing the SVM classification and SVM regression underlying mathematics as massively parallelised FPGA-based DSP pipeline hardware, and, applying these systems and software models to several distinct application domains.

A brute-force search classification training architecture has been successfully modelled and verified. Additionally two different classes of ANN-mapped optimisation strategies have also been proposed for SVM classification and regression. Each proposed training strategy was designed as parallelised architectural-structures and functional-blocks suitable for hardware implementation.

A suite of FPGA-based SVM hardware and accompanying tools, models, and software has been developed, implemented, and applied throughout the scope this work. The systems implemented include the SVM classification and SVM regression DSP pipelines, a hardware test-rig and ancillary software subsystems, and, SVM training and function evaluation DSP pipeline software models.

The SVM classification training DSP pipeline architecture implementations are compatible with data-sets spanning 2 to 8 dimensions and support vector sets of up to 16 support vectors. The SVM classification training pipeline architecture implementations have a minimum pipeline latency of 0.18 microseconds and a maximum pipeline latency of 0.28 microseconds.

The SVM classification function evaluation DSP pipeline architecture implementations as part of this work are compatible with data-sets spanning 2 to 8 dimensions and support vector sets of up to 32 support vectors. The SVM classification function evaluation pipeline architecture implementations have a minimum pipeline latency of 0.16 microsec-

onds and a maximum pipeline latency of 0.24 microseconds.

The SVM regression training DSP pipeline architecture implementations as part of this work are compatible with data-sets spanning 2 to 8 dimensions and support vector sets of up to 16 support vectors. The SVM regression training pipeline architecture implementations have a minimum pipeline latency of 0.20 microseconds and a maximum pipeline latency of 0.30 microseconds.

The SVM regression function evaluation DSP pipeline architecture implementations as part of this work are compatible with data-sets spanning 2 to 8 dimensions and support vector sets of up to 16 support vectors. The SVM regression function evaluation pipeline architecture implementations have a minimum pipeline latency of 0.20 microseconds and a maximum pipeline latency of 0.30 microseconds.

Utilising LIBSVM training and the parallelised SVM DSP pipeline function evaluation architecture prototypes SVM classification and SVM regression was successfully applied to Rajkumar's oil and gas pipeline fault detection and failure system legacy data-set yielding excellent results.

Finally, also utilising LIBSVM training, and, the parallelised SVM DSP pipeline function evaluation architecture prototypes, SVM classification and SVM regression was also applied to several chaotic systems as a feasibility study into the application of the SVM machine learning paradigm in nonlinear and chaotic dynamical systems domain. SVM classification was applied to the Lorenz Attractor and an ANN-based chaotic oscillator to a reasonably acceptable degree of success. SVM classification was applied to the Mackey-Glass attractor yielding poor results. SVM regression was applied Lorenz Attractor and an ANN-based chaotic oscillator yielding average but encouraging results. SVM regression was applied to the Mackey-Glass attractor yielding poor results.

# Appendices

## Appendix A.

## SVM DSP Instruction Set

Each DSP pipeline consists of a series of carefully mapped parallel mathematical operations. Table A.1 to Table A.7 provide an overview of each mathematical operation utilised throughout each DSP pipeline. Note that these operations are generalised for the sake of both documentation and VHDL code clarity.

***Table A.1:*** *Linear Kernel Specific DSP Instructions.*

| Code | Instruction Overview |
|---|---|
| `kmd0.` | Linear kernel matrix $\boldsymbol{\psi}_{ij}$ Construction: $i \times j$ parallel `dot0.` ops. |
| `kvd0.` | Linear kernel vector $\bar{\boldsymbol{\psi}}_i$ Construction: $i$ parallel `dot0.` ops. |

***Table A.2:*** *Polynomial Kernel Specific DSP Instructions.*

| Code | Instruction Overview |
|---|---|
| `kmp2.` | Polynomial kernel matrix $\boldsymbol{\psi}_{ij}$ Construction: $i \times j$ parallel `dot0.` ops. |
| `kmp1.` | Polynomial kernel matrix $\boldsymbol{\psi}_{ij}$ Construction: $+\, \boldsymbol{c}_{i \times j}$ matrix. |
| `kmp0.` | Polynomial kernel matrix $\boldsymbol{\psi}_{ij}$ Construction: squared ops. |
| `kvp2.` | Polynomial kernel vector $\bar{\boldsymbol{\psi}}_i$ Construction: $i$ parallel `dot0.` ops. |
| `kvp1.` | Polynomial kernel vector $\bar{\boldsymbol{\psi}}_i$ Construction: $+\, \bar{\boldsymbol{c}}_i$ vector. |
| `kvp0.` | Polynomial kernel vector $\bar{\boldsymbol{\psi}}_i$ Construction: squared ops. |

***Table A.3:*** *Gaussian Kernel Specific DSP Instructions.*

| Code | Instruction Overview |
|---|---|
| `kmg3.` | Gaussian kernel matrix $\boldsymbol{\psi}_{ij}$: $i \times j \times d$ parallel `sub0.` ops. |
| `kmg2.` | Gaussian kernel matrix $\boldsymbol{\psi}_{ij}$ Construction: $i \times j$ parallel `dot0.` ops. |
| `kmg1.` | Gaussian kernel matrix $\boldsymbol{\psi}_{ij}$ Construction: $\div\, 2\sigma^2$ ops. |
| `kmg0.` | Gaussian kernel matrix $\boldsymbol{\psi}_{ij}$ Construction: parallel `tse0.` ops. |
| `kvg3.` | Gaussian kernel vector $\bar{\boldsymbol{\psi}}_i$ Construction: $i \times d$ parallel `sub0.` ops. |
| `kvg2.` | Gaussian kernel vector $\bar{\boldsymbol{\psi}}_i$ Construction: $i$ parallel `dot0.` ops. |
| `kvg1.` | Gaussian kernel vector $\bar{\boldsymbol{\psi}}_i$ Construction: $\div\, 2\sigma^2$ ops. |
| `kvg0.` | Gaussian kernel vector $\bar{\boldsymbol{\psi}}_i$ Construction: parallel `tse0.` ops. |

***Table A.4:*** *Radial Basis Function (RBF) Kernel Specific DSP Instructions.*

| Code | Instruction Overview |
|---|---|
| `kmr3.` | RBF kernel matrix $\boldsymbol{\psi}_{ij}$: $i \times j \times d$ parallel `sub0.` ops. |
| `kmr2.` | RBF kernel matrix $\boldsymbol{\psi}_{ij}$ Construction: $i \times j$ parallel `dot0.` ops. |
| `kmr1.` | RBF kernel matrix $\boldsymbol{\psi}_{ij}$ Construction: $\times \gamma$ ops. |
| `kmr0.` | RBF kernel matrix $\boldsymbol{\psi}_{ij}$ Construction: parallel `tse0.` ops. |
| `kvr3.` | RBF kernel vector $\bar{\boldsymbol{\psi}}_i$ Construction: $i \times d$ parallel `sub0.` ops. |
| `kvr2.` | RBF kernel vector $\bar{\boldsymbol{\psi}}_i$ Construction: $i$ parallel `dot0.` ops. |
| `kvr1.` | RBF kernel vector $\bar{\boldsymbol{\psi}}_i$ Construction: $\times \gamma$ ops. |
| `kvr0.` | RBF kernel vector $\bar{\boldsymbol{\psi}}_i$ Construction: parallel `tse0.` ops. |

***Table A.5:*** *Sigmoid / Hyperbolic Tangent Kernel Specific DSP Instructions.*

| Code | Instruction Overview |
|---|---|
| `kmh2.` | Polynomial kernel matrix $\boldsymbol{\psi}_{ij}$ Construction: $i \times j$ parallel `dot0.` ops. |
| `kmh1.` | Polynomial kernel matrix $\boldsymbol{\psi}_{ij}$ Construction: $+ \boldsymbol{c}_{i \times j}$ matrix. |
| `kmh0.` | Polynomial kernel matrix $\boldsymbol{\psi}_{ij}$ Construction: tanh ops. |
| `kvh2.` | Polynomial kernel vector $\bar{\boldsymbol{\psi}}_i$ Construction: $i$ parallel `dot0.` ops. |
| `kvh1.` | Polynomial kernel vector $\bar{\boldsymbol{\psi}}_i$ Construction: $+ \bar{\boldsymbol{c}}_i$ vector. |
| `kvh0.` | Polynomial kernel vector $\bar{\boldsymbol{\psi}}_i$ Construction: tanh ops. |

***Table A.6:*** *Generic DSP Instructions.*

| Code | Instruction Overview |
|---|---|
| `add0.` | Add two scalar values. |
| `sub0.` | Subtract two scalar values. |
| `sum0.` | Adder tree: sum of $i$ scalar values. |
| `sys0.` | Systolic sum pipeline: sum of $i$ scalar values. |
| `mult0.` | Multiply two scalar values. |
| `dot0.` | Dot product of two $\bar{\boldsymbol{x}}$ vectors. |
| `dot1.` | Dot product of a vector and a scalar. |
| `dot2.` | Dot product of $\bar{\boldsymbol{y}}$ and $\bar{\boldsymbol{\alpha}}$ vectors. |
| `tse0.` | Taylor Series exponential function $e^x$ approximation. |
| `tsh0.` | Taylor Series hyperbolic tangent function $\tanh(x)$ approximation. |
| `ayv0.` | $\bar{\boldsymbol{\alpha}}_i$ or values $\times \bar{\boldsymbol{y}}_i$ values vector construction. |
| `aat0.` | Matrix construction: $\bar{\boldsymbol{\alpha}}_{ij}$ or $\bar{\boldsymbol{a}}_{ij}$. |
| `an0.` | Regression: $\bar{\boldsymbol{\alpha}} - \bar{\boldsymbol{\alpha}}$. |
| `ap0.` | Regression: $\bar{\boldsymbol{\alpha}} + \bar{\boldsymbol{\alpha}}$. |
| `ea1.` | Sum of $\bar{\boldsymbol{\alpha}}$ vector. |
| `ea0.` | Scalar multiplication $\alpha \times \epsilon$. |

***Table A.7:*** *Pipeline-specific DSP Instruction Set.*

| `ct0.` Pipeline Classification Training Instructions | |
|---|---|
| Code | Instruction Overview |
| `ctlhs0.` | Sum of $\bar{\boldsymbol{\alpha}}$ vector. |
| `ctrhs2.` | Matrix construction: $\boldsymbol{a}\boldsymbol{a}_{ij}^T \times \boldsymbol{\psi}_{ij}$. |
| `ctrhs1.` | Sum of rows. |
| `ctrhs0.` | Sum of columns. |
| `ct0.` | `ctlhs0.` $-$ `ctrhs0.` |

| `ce0.` Pipeline Classification Evaluation Instructions | |
|---|---|
| Code | Instruction Overview |
| `cerhs0.` | Dot product: $\bar{\boldsymbol{a}}^T\bar{\boldsymbol{\psi}}_{\bar{\boldsymbol{x}}} = \bar{\boldsymbol{a}} \bullet \bar{\boldsymbol{\psi}}_{\bar{\boldsymbol{x}}}$. |
| `celhsp1.` | Dot product: $\bar{\boldsymbol{a}}^T\bar{\boldsymbol{\psi}}_{+} = \bar{\boldsymbol{a}} \bullet \bar{\boldsymbol{\psi}}_{+}$. |
| `celhsn1.` | Dot product: $\bar{\boldsymbol{a}}^T\bar{\boldsymbol{\psi}}_{-} = \bar{\boldsymbol{a}} \bullet \bar{\boldsymbol{\psi}}_{-}$. |
| `celhs0.` | `celhsp1.` + `celhsn1.` |
| `ce0.` | `cerhs0.` $-$ `celhs0.` |

| `rt0.` Pipeline Regression Training Instructions | |
|---|---|
| Code | Instruction Overview |
| `rtlhs0.` | Scalar $-$ scalar. |
| `rtrhs2.` | Matrix construction: $\bar{\boldsymbol{\alpha}}\bar{\boldsymbol{\alpha}}_{ij}^T \times \boldsymbol{\psi}_{ij}$. |
| `rtrhs1.` | Sum of rows. |
| `rtrhs0.` | Sum of columns. |
| `rt0.` | `rtlhs0.` $-$ `rtrhs0.` |

| `re0.` Pipeline Regression Evaluation Instructions | |
|---|---|
| Code | Instruction Overview |
| `relhs0.` | Dot product: $\bar{\boldsymbol{\alpha}}^T\bar{\boldsymbol{\psi}} = \bar{\boldsymbol{\alpha}} \bullet \bar{\boldsymbol{\psi}}$. |
| `rerhs3.` | Matrix construction: $\boldsymbol{\alpha}_{ij}^T \times \boldsymbol{\psi}_{ij}$. |
| `rerhs2.` | Matrix construction: $-\,\boldsymbol{y}_{ij}$ matrix. |
| `rerhs1.` | Sum of rows. |
| `rerhs0.` | Sum of columns. |
| `re0.` | `relhs0.` $-$ `rerhs0.` |

# Appendix B. Kernel Pipeline Designs

Figure A.1 illustrates the Gaussian kernel as a pipeline of staged-instructions. Table A.8 lists the Gaussian kernel pipeline instruction set and corresponding operations.

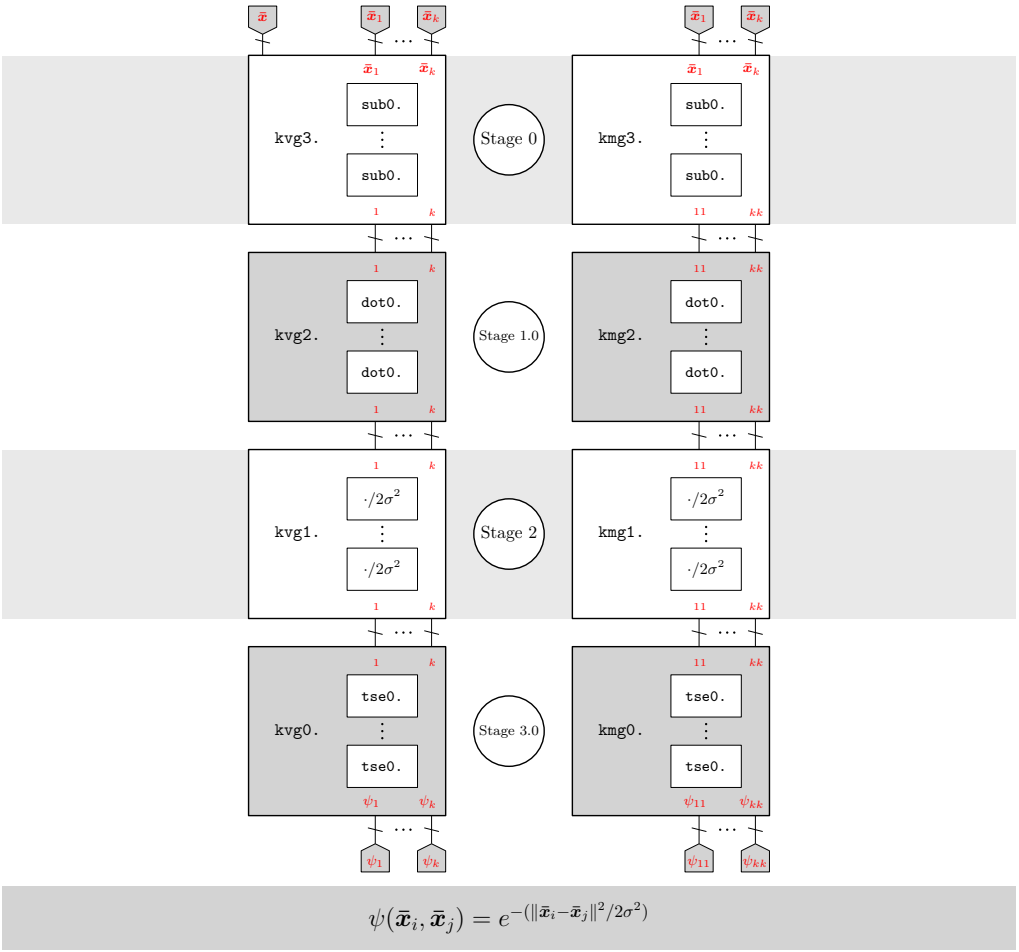



***Figure A.1:*** *Gaussian kernel pipeline.*

The Gaussian kernel pipeline exponential operation instructions `kvg0.` and `kmg0.` shown in Fig. A.1 and Table A.8 can be as a finite power series approximation.

***Table A.8:*** *Gaussian kernel pipeline instruction overview.*

| Instruction | Mathematical Operation |
|---|---|
| | $\psi(\bar{\boldsymbol{x}}_i, \bar{\boldsymbol{x}}) \Rightarrow \bar{\boldsymbol{\psi}}_{i\times 1} = \bar{\boldsymbol{\psi}}$ |
| `kvg3.` | $\Rightarrow \begin{bmatrix} \bar{\boldsymbol{x}}_1 - \bar{\boldsymbol{x}} \\ \vdots \\ \bar{\boldsymbol{x}}_k - \bar{\boldsymbol{x}} \end{bmatrix}$ |
| `kvg2.` | $\Rightarrow \begin{bmatrix} \lVert\bar{\boldsymbol{x}}_1 - \bar{\boldsymbol{x}}\rVert^2 \\ \vdots \\ \lVert\bar{\boldsymbol{x}}_k - \bar{\boldsymbol{x}}\rVert^2 \end{bmatrix} = \begin{bmatrix} \bar{\boldsymbol{a}}_1 \cdot \bar{\boldsymbol{a}}_1 \\ \vdots \\ \bar{\boldsymbol{a}}_k \cdot \bar{\boldsymbol{a}}_k \end{bmatrix}$ |
| `kvg1.` | $\Rightarrow \begin{bmatrix} \lVert\bar{\boldsymbol{x}}_1 - \bar{\boldsymbol{x}}\rVert^2/2\sigma^2 \\ \vdots \\ \lVert\bar{\boldsymbol{x}}_k - \bar{\boldsymbol{x}}\rVert^2/2\sigma^2 \end{bmatrix}$ |
| `kvg0.` | $\Rightarrow \begin{bmatrix} e^{-(\lVert\bar{\boldsymbol{x}}_1 - \bar{\boldsymbol{x}}\rVert^2/2\sigma^2)} \\ \vdots \\ e^{-(\lVert\bar{\boldsymbol{x}}_k - \bar{\boldsymbol{x}}\rVert^2/2\sigma^2)} \end{bmatrix} = \begin{bmatrix} \psi_1 \\ \vdots \\ \psi_k \end{bmatrix} = \bar{\boldsymbol{\psi}}$ |
| | $\psi(\bar{\boldsymbol{x}}_i, \bar{\boldsymbol{x}}_j) \Rightarrow \boldsymbol{\psi}_{i\times j} = \boldsymbol{\psi}$ |
| `kmg3.` | $\Rightarrow \begin{bmatrix} \bar{\boldsymbol{x}}_1 - \bar{\boldsymbol{x}}_1 & \cdots & \bar{\boldsymbol{x}}_1 - \bar{\boldsymbol{x}}_k \\ \vdots & \ddots & \vdots \\ \bar{\boldsymbol{x}}_k - \bar{\boldsymbol{x}}_1 & \cdots & \bar{\boldsymbol{x}}_k - \bar{\boldsymbol{x}}_k \end{bmatrix}$ |
| `kmg2.` | $\Rightarrow \begin{bmatrix} \lVert\bar{\boldsymbol{x}}_1 - \bar{\boldsymbol{x}}_1\rVert^2 & \cdots & \lVert\bar{\boldsymbol{x}}_1 - \bar{\boldsymbol{x}}_k\rVert^2 \\ \vdots & \ddots & \vdots \\ \lVert\bar{\boldsymbol{x}}_k - \bar{\boldsymbol{x}}_1\rVert^2 & \cdots & \lVert\bar{\boldsymbol{x}}_k - \bar{\boldsymbol{x}}_k\rVert^2 \end{bmatrix} = \begin{bmatrix} \bar{\boldsymbol{a}}_1 \cdot \bar{\boldsymbol{a}}_1 & \cdots & \bar{\boldsymbol{a}}_1 \cdot \bar{\boldsymbol{a}}_k \\ \vdots & \ddots & \vdots \\ \bar{\boldsymbol{a}}_k \cdot \bar{\boldsymbol{a}}_1 & \cdots & \bar{\boldsymbol{a}}_k \cdot \bar{\boldsymbol{a}}_k \end{bmatrix}$ |
| `kmg1.` | $\Rightarrow \begin{bmatrix} \lVert\bar{\boldsymbol{x}}_1 - \bar{\boldsymbol{x}}_1\rVert^2/2\sigma^2 & \cdots & \lVert\bar{\boldsymbol{x}}_1 - \bar{\boldsymbol{x}}_k\rVert^2/2\sigma^2 \\ \vdots & \ddots & \vdots \\ \lVert\bar{\boldsymbol{x}}_k - \bar{\boldsymbol{x}}_1\rVert^2/2\sigma^2 & \cdots & \lVert\bar{\boldsymbol{x}}_k - \bar{\boldsymbol{x}}_k\rVert^2/2\sigma^2 \end{bmatrix}$ |
| `kmg0.` | $\Rightarrow \begin{bmatrix} e^{-(\lVert\bar{\boldsymbol{x}}_1 - \bar{\boldsymbol{x}}_1\rVert^2/2\sigma^2)} & \cdots & e^{-(\lVert\bar{\boldsymbol{x}}_1 - \bar{\boldsymbol{x}}_k\rVert^2/2\sigma^2)} \\ \vdots & \ddots & \vdots \\ e^{-(\lVert\bar{\boldsymbol{x}}_k - \bar{\boldsymbol{x}}_1\rVert^2/2\sigma^2)} & \cdots & e^{-(\lVert\bar{\boldsymbol{x}}_k - \bar{\boldsymbol{x}}_k\rVert^2/2\sigma^2)} \end{bmatrix} = \begin{bmatrix} \boldsymbol{\psi}_{11} & \cdots & \boldsymbol{\psi}_{1k} \\ \vdots & \ddots & \vdots \\ \boldsymbol{\psi}_{k1} & \cdots & \boldsymbol{\psi}_{kk} \end{bmatrix} = \boldsymbol{\psi}$ |

Figure A.2 illustrates the Radial Basis Function (RBF) kernel as a pipeline of staged-instructions. Table A.9 lists the Radial Basis Function (RBF) kernel pipeline instruction set and corresponding operations.

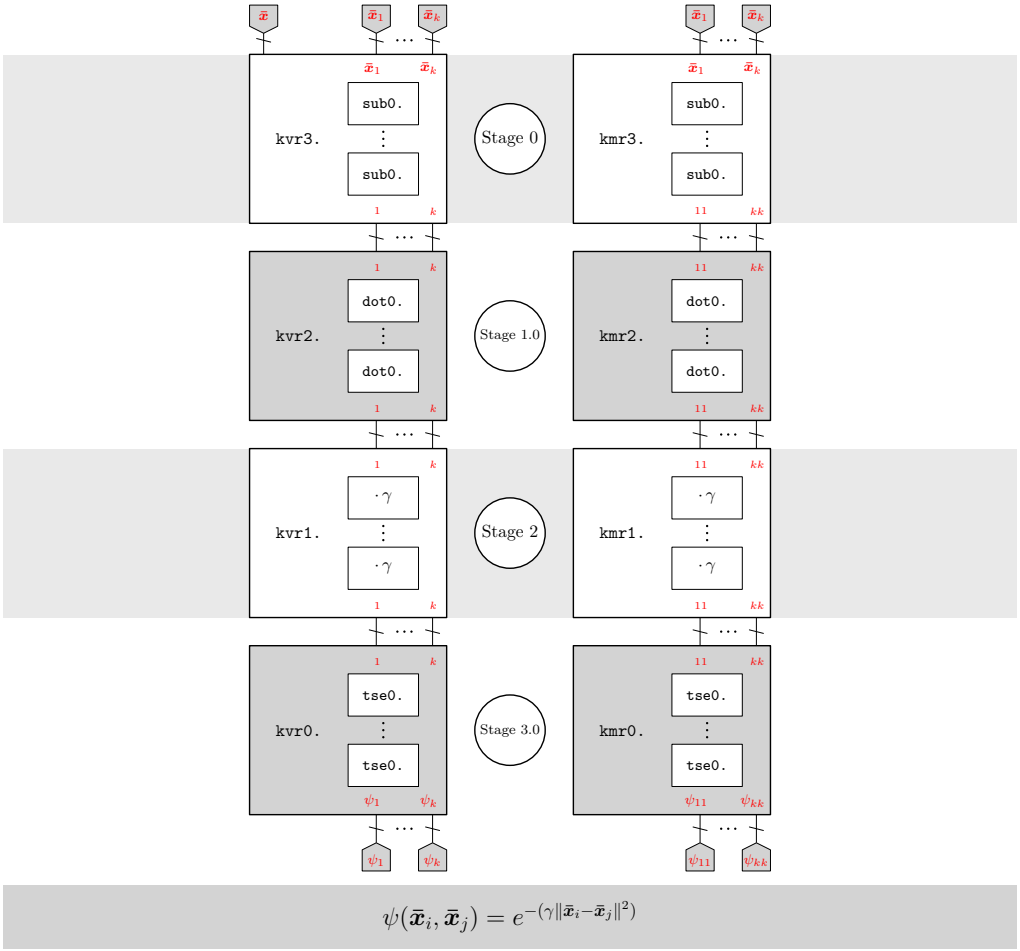



***Figure A.2:*** *Radial Basis Function (RBF) kernel pipeline.*

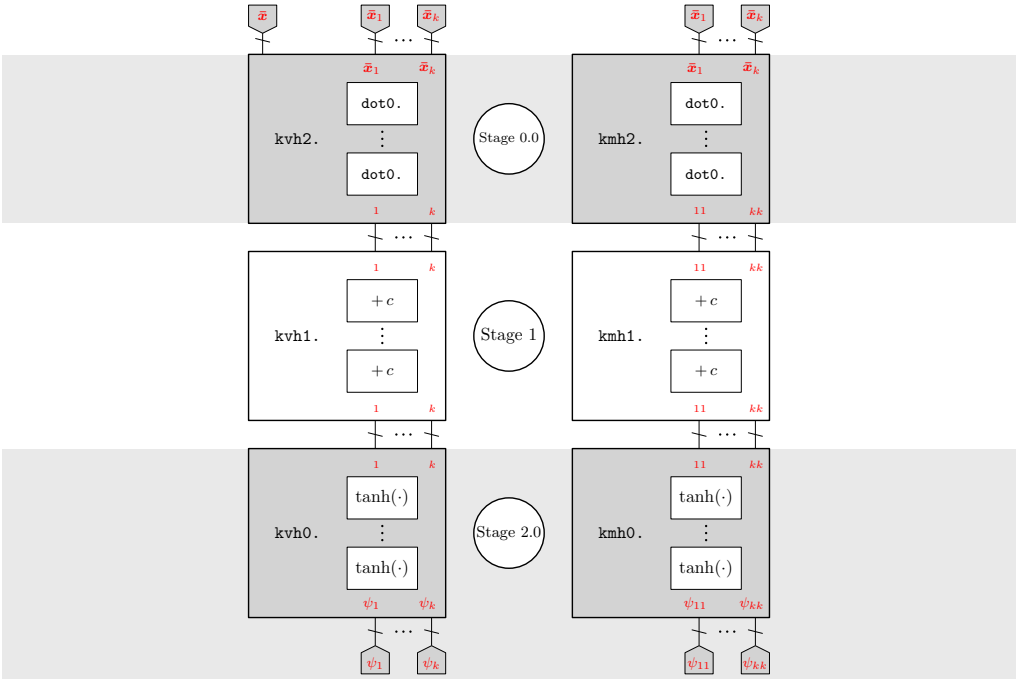
The Radial Basis Function kernel pipeline exponential operation instructions `kvr0.` and `kmr0.` shown in Fig. A.2 and Table A.9 can be as a finite power series approximation.

***Table A.9:*** *Radial Basis Function (RBF) kernel pipeline instruction overview.*

| Instruction | Mathematical Operation |
|---|---|
| | $\psi(\bar{\boldsymbol{x}}_i, \bar{\boldsymbol{x}}) \Rightarrow \bar{\boldsymbol{\psi}}_{i\times 1} = \bar{\boldsymbol{\psi}}$ |
| `kvg3.` | $\Rightarrow \begin{bmatrix} \bar{\boldsymbol{x}}_1 - \bar{\boldsymbol{x}} \\ \vdots \\ \bar{\boldsymbol{x}}_k - \bar{\boldsymbol{x}} \end{bmatrix}$ |
| `kvg2.` | $\Rightarrow \begin{bmatrix} \lVert\bar{\boldsymbol{x}}_1 - \bar{\boldsymbol{x}}\rVert^2 \\ \vdots \\ \lVert\bar{\boldsymbol{x}}_k - \bar{\boldsymbol{x}}\rVert^2 \end{bmatrix} = \begin{bmatrix} \bar{\boldsymbol{a}}_1 \cdot \bar{\boldsymbol{a}}_1 \\ \vdots \\ \bar{\boldsymbol{a}}_k \cdot \bar{\boldsymbol{a}}_k \end{bmatrix}$ |
| `kvg1.` | $\Rightarrow \begin{bmatrix} \gamma\lVert\bar{\boldsymbol{x}}_1 - \bar{\boldsymbol{x}}\rVert^2 \\ \vdots \\ \gamma\lVert\bar{\boldsymbol{x}}_k - \bar{\boldsymbol{x}}\rVert^2 \end{bmatrix}$ |
| `kvg0.` | $\Rightarrow \begin{bmatrix} e^{-(\gamma\lVert\bar{\boldsymbol{x}}_1 - \bar{\boldsymbol{x}}\rVert^2)} \\ \vdots \\ e^{-(\gamma\lVert\bar{\boldsymbol{x}}_k - \bar{\boldsymbol{x}}\rVert^2)} \end{bmatrix} = \begin{bmatrix} \psi_1 \\ \vdots \\ \psi_k \end{bmatrix} = \bar{\boldsymbol{\psi}}$ |
| | $\psi(\bar{\boldsymbol{x}}_i, \bar{\boldsymbol{x}}_j) \Rightarrow \boldsymbol{\psi}_{i\times j} = \boldsymbol{\psi}$ |
| `kmg3.` | $\Rightarrow \begin{bmatrix} \bar{\boldsymbol{x}}_1 - \bar{\boldsymbol{x}}_1 & \cdots & \bar{\boldsymbol{x}}_1 - \bar{\boldsymbol{x}}_k \\ \vdots & \ddots & \vdots \\ \bar{\boldsymbol{x}}_k - \bar{\boldsymbol{x}}_1 & \cdots & \bar{\boldsymbol{x}}_k - \bar{\boldsymbol{x}}_k \end{bmatrix}$ |
| `kmg2.` | $\Rightarrow \begin{bmatrix} \lVert\bar{\boldsymbol{x}}_1 - \bar{\boldsymbol{x}}_1\rVert^2 & \cdots & \lVert\bar{\boldsymbol{a}}_1 - \bar{\boldsymbol{a}}_k\rVert^2 \\ \vdots & \ddots & \vdots \\ \lVert\bar{\boldsymbol{x}}_k - \bar{\boldsymbol{x}}_1\rVert^2 & \cdots & \lVert\bar{\boldsymbol{x}}_k - \bar{\boldsymbol{x}}_k\rVert^2 \end{bmatrix} = \begin{bmatrix} \bar{\boldsymbol{a}}_1 \cdot \bar{\boldsymbol{a}}_1 & \cdots & \bar{\boldsymbol{a}}_1 \cdot \bar{\boldsymbol{a}}_k \\ \vdots & \ddots & \vdots \\ \bar{\boldsymbol{a}}_k \cdot \bar{\boldsymbol{a}}_1 & \cdots & \bar{\boldsymbol{a}}_k \cdot \bar{\boldsymbol{a}}_k \end{bmatrix}$ |
| `kmg1.` | $\Rightarrow \begin{bmatrix} \gamma\lVert\bar{\boldsymbol{x}}_1 - \bar{\boldsymbol{x}}_1\rVert^2 & \cdots & \gamma\lVert\bar{\boldsymbol{x}}_1 - \bar{\boldsymbol{x}}_k\rVert^2 \\ \vdots & \ddots & \vdots \\ \gamma\lVert\bar{\boldsymbol{x}}_k - \bar{\boldsymbol{x}}_1\rVert^2 & \cdots & \gamma\lVert\bar{\boldsymbol{x}}_k - \bar{\boldsymbol{x}}_k\rVert^2 \end{bmatrix}$ |
| `kmg0.` | $\Rightarrow \begin{bmatrix} e^{-(\gamma\lVert\bar{\boldsymbol{x}}_1 - \bar{\boldsymbol{x}}_1\rVert^2)} & \cdots & e^{-(\gamma\lVert\bar{\boldsymbol{x}}_1 - \bar{\boldsymbol{x}}_k\rVert^2)} \\ \vdots & \ddots & \vdots \\ e^{-(\gamma\lVert\bar{\boldsymbol{x}}_k - \bar{\boldsymbol{x}}_1\rVert^2)} & \cdots & e^{-(\gamma\lVert\bar{\boldsymbol{x}}_k - \bar{\boldsymbol{x}}_k\rVert^2)} \end{bmatrix} = \begin{bmatrix} \boldsymbol{\psi}_{11} & \cdots & \boldsymbol{\psi}_{1k} \\ \vdots & \ddots & \vdots \\ \boldsymbol{\psi}_{k1} & \cdots & \boldsymbol{\psi}_{kk} \end{bmatrix} = \boldsymbol{\psi}$ |

Figure A.3 illustrates the Sigmoid / Hyperbolic Tangent kernel as a pipeline of staged-instructions. Table A.10 lists the Sigmoid / Hyperbolic Tangent kernel pipeline instruction set and corresponding operations.

The Sigmoid / Hyperbolic Tangent kernel pipeline hyperbolic-tangent operation instructions `kvh0.` and `kmh0.` shown in Fig. A.3 and Table A.10 can be as a finite Taylor series approximation.




$$\psi(\bar{\boldsymbol{x}}_i, \bar{\boldsymbol{x}}_j) = \tanh(\bar{\boldsymbol{x}}_i \bullet \bar{\boldsymbol{x}}_j + c)$$

***Figure A.3:*** *Sigmoid / Hyperbolic Tangent Kernel Pipeline.*

***Table A.10:*** *Sigmoid / Hyperbolic Tangent kernel pipeline instruction overview.*

| Instruction | Mathematical Operation |
|---|---|
| | $\psi(\bar{\boldsymbol{x}}_i, \bar{\boldsymbol{x}}) \Rightarrow \bar{\boldsymbol{\psi}}_{i\times 1} = \bar{\boldsymbol{\psi}}$ |
| kvh2. | $\Rightarrow \begin{bmatrix} \bar{\boldsymbol{x}}_1 \cdot \bar{\boldsymbol{x}} \\ \vdots \\ \bar{\boldsymbol{x}}_k \cdot \bar{\boldsymbol{x}} \end{bmatrix}$ |
| kvh1. | $\Rightarrow \begin{bmatrix} [\bar{\boldsymbol{x}}_1 \cdot \bar{\boldsymbol{x}}] + c \\ \vdots \\ [\bar{\boldsymbol{x}}_k \cdot \bar{\boldsymbol{x}}] + c \end{bmatrix}$ |
| kvh0. | $\Rightarrow \begin{bmatrix} \tanh([\bar{\boldsymbol{x}}_1 \cdot \bar{\boldsymbol{x}}] + c) \\ \vdots \\ \tanh([\bar{\boldsymbol{x}}_k \cdot \bar{\boldsymbol{x}}] + c) \end{bmatrix} = \begin{bmatrix} \psi_1 \\ \vdots \\ \psi_k \end{bmatrix} = \bar{\boldsymbol{\psi}}$ |
| | $\psi(\bar{\boldsymbol{x}}_i, \bar{\boldsymbol{x}}_j) \Rightarrow \boldsymbol{\psi}_{i\times j} = \boldsymbol{\psi}$ |
| kmh2. | $\Rightarrow \begin{bmatrix} \bar{\boldsymbol{x}}_1 \cdot \bar{\boldsymbol{x}}_1 & \cdots & \bar{\boldsymbol{x}}_1 \cdot \bar{\boldsymbol{x}}_k \\ \vdots & \ddots & \vdots \\ \bar{\boldsymbol{x}}_k \cdot \bar{\boldsymbol{x}}_1 & \cdots & \bar{\boldsymbol{x}}_k \cdot \bar{\boldsymbol{x}}_k \end{bmatrix}$ |
| kmh1. | $\Rightarrow \begin{bmatrix} [\bar{\boldsymbol{x}}_1 \cdot \bar{\boldsymbol{x}}_1] + c & \cdots & [\bar{\boldsymbol{x}}_1 \cdot \bar{\boldsymbol{x}}_k] + c \\ \vdots & \ddots & \vdots \\ [\bar{\boldsymbol{x}}_k \cdot \bar{\boldsymbol{x}}_1] + c & \cdots & [\bar{\boldsymbol{x}}_k \cdot \bar{\boldsymbol{x}}_k] + c \end{bmatrix}$ |
| kmh0. | $\Rightarrow \begin{bmatrix} \tanh([\bar{\boldsymbol{x}}_1 \cdot \bar{\boldsymbol{x}}_1] + c) & \cdots & \tanh([\bar{\boldsymbol{x}}_1 \cdot \bar{\boldsymbol{x}}_k] + c) \\ \vdots & \ddots & \vdots \\ \tanh([\bar{\boldsymbol{x}}_k \cdot \bar{\boldsymbol{x}}_1] + c) & \cdots & \tanh([\bar{\boldsymbol{x}}_k \cdot \bar{\boldsymbol{x}}_k] + c) \end{bmatrix} = \begin{bmatrix} \boldsymbol{\psi}_{11} & \cdots & \boldsymbol{\psi}_{1k} \\ \vdots & \ddots & \vdots \\ \boldsymbol{\psi}_{k1} & \cdots & \boldsymbol{\psi}_{kk} \end{bmatrix} = \boldsymbol{\psi}$ |

## Appendix C. Implemented Pipeline Entities

Listing A.1 provides the VHDL Entity for `dsp_d2_k4_ct0`. Classification Evaluation Pipeline. Listing A.2 provides the VHDL Entity for `dsp_d2_k8_ct0`. Classification Evaluation Pipeline. Listing A.3 provides the VHDL Entity for `dsp_d2_k16_ct0`. Classification Evaluation Pipeline. Listing A.4 provides the VHDL Entity for `dsp_d2_k32_ct0`. Classification Evaluation Pipeline.

***Listing A.1:*** *VHDL Entity:* `dsp_d2_k4_ct0`. *Classification Evaluation Pipeline.*

```
1  -- dsp_d2_k4_ct0 Entity
2  ENTITY dsp_d2_k4_ct0 IS
3  GENERIC
4  (
5      CT0_D_C   : NATURAL := 2;
6      CT0_K_C   : NATURAL := 4;
7      CT0_ST_C  : NATURAL := 9
8  );
9  PORT
10 (
11     -- Control io
12     clk       : IN  STD_LOGIC;
13     rst       : IN  STD_LOGIC;
14     en        : IN  STD_LOGIC;
15     valid     : OUT STD_LOGIC;
16     -- Input Signals
17     xv        : IN  CT0_X_ARRAY;
18     a         : IN  CT0_A_TYPE;
19     y         : IN  CT0_Y_TYPE;
20     -- Output.
21     ct0       : OUT CT0_TYPE
22 );
23 END dsp_d2_k4_ct0;
```

***Listing A.2:*** *VHDL Entity:* `dsp_d2_k8_ct0`. *Classification Evaluation Pipeline.*

```
1  -- dsp_d2_k8_ct0 Entity
2  ENTITY dsp_d2_k8_ct0 IS
3  GENERIC
4  (
5      CT0_D_C   : NATURAL := 2;
6      CT0_K_C   : NATURAL := 8;
7      CT0_ST_C  : NATURAL := 11
8  );
9  PORT
10 (
11     -- Control io
12     clk       : IN  STD_LOGIC;
13     rst       : IN  STD_LOGIC;
14     en        : IN  STD_LOGIC;
15     valid     : OUT STD_LOGIC;
16     -- Input Signals
17     xv        : IN  CT0_X_ARRAY;
18     a         : IN  CT0_A_TYPE;
19     y         : IN  CT0_Y_TYPE;
20     -- Output.
21     ct0       : OUT CT0_TYPE
22 );
23 END dsp_d2_k8_ct0;
```

***Listing A.3:*** *VHDL Entity:* `dsp_d2_k16_ct0`. *Classification Evaluation Pipeline.*

```
-- dsp_d2_k16_ct0 Entity
ENTITY dsp_d2_k16_ct0 IS
GENERIC
(
    CT0_D_C   : NATURAL := 2;
    CT0_K_C   : NATURAL := 16;
    CT0_ST_C  : NATURAL := 13
);
PORT
(
    -- Control io
    clk       : IN  STD_LOGIC;
    rst       : IN  STD_LOGIC;
    en        : IN  STD_LOGIC;
    valid     : OUT STD_LOGIC;
    -- Input Signals
    xv        : IN  CT0_X_ARRAY;
    a         : IN  CT0_A_TYPE;
    y         : IN  CT0_Y_TYPE;
    -- Output.
    ct0       : OUT CT0_TYPE
);
END dsp_d2_k16_ct0;
```

***Listing A.4:*** *VHDL Entity:* `dsp_d2_k32_ct0`. *Classification Evaluation Pipeline.*

```
-- dsp_d2_k32_ct0 Entity
ENTITY dsp_d2_k32_ct0 IS
GENERIC
(
    CT0_D_C   : NATURAL := 2;
    CT0_K_C   : NATURAL := 32;
    CT0_ST_C  : NATURAL := 15
);
PORT
(
    -- Control io
    clk       : IN  STD_LOGIC;
    rst       : IN  STD_LOGIC;
    en        : IN  STD_LOGIC;
    valid     : OUT STD_LOGIC;
    -- Input Signals
    xv        : IN  CT0_X_ARRAY;
    a         : IN  CT0_A_TYPE;
    y         : IN  CT0_Y_TYPE;
    -- Output.
    ct0       : OUT CT0_TYPE
);
END dsp_d2_k32_ct0;
```

Listing A.5 provides the VHDL Entity for `dsp_d4_k8_ct0`. Classification Evaluation Pipeline. Listing A.6 provides the VHDL Entity for `dsp_d4_k16_ct0`. Classification Evaluation Pipeline. Listing A.7 provides the VHDL Entity for `dsp_d4_k32_ct0`. Classification Evaluation Pipeline.

***Listing A.5:*** *VHDL Entity:* `dsp_d4_k8_ct0`. *Classification Evaluation Pipeline.*

```
-- dsp_d4_k8_ct0 Entity
ENTITY dsp_d4_k8_ct0 IS
GENERIC
(
    CT0_D_C   : NATURAL := 4;
    CT0_K_C   : NATURAL := 8;
    CT0_ST_C  : NATURAL := 11
);
PORT
(
    -- Control io
    clk       : IN  STD_LOGIC;
    rst       : IN  STD_LOGIC;
    en        : IN  STD_LOGIC;
    valid     : OUT STD_LOGIC;
    -- Input Signals
    xv        : IN  CT0_X_ARRAY;
    a         : IN  CT0_A_TYPE;
    y         : IN  CT0_Y_TYPE;
    -- Output.
    ct0       : OUT CT0_TYPE
);
END dsp_d4_k8_ct0;
```

***Listing A.6:*** *VHDL Entity:* `dsp_d4_k16_ct0`. *Classification Evaluation Pipeline.*

```
-- dsp_d4_k16_ct0 Entity
ENTITY dsp_d4_k16_ct0 IS
GENERIC
(
    CT0_D_C   : NATURAL := 4;
    CT0_K_C   : NATURAL := 16;
    CT0_ST_C  : NATURAL := 13
);
PORT
(
    -- Control io
    clk       : IN  STD_LOGIC;
    rst       : IN  STD_LOGIC;
    en        : IN  STD_LOGIC;
    valid     : OUT STD_LOGIC;
    -- Input Signals
    xv        : IN  CT0_X_ARRAY;
    a         : IN  CT0_A_TYPE;
    y         : IN  CT0_Y_TYPE;
    -- Output.
    ct0       : OUT CT0_TYPE
);
END dsp_d4_k16_ct0;
```

***Listing A.7:*** *VHDL Entity:* `dsp_d4_k32_ct0`. *Classification Evaluation Pipeline.*

```
-- dsp_d4_k32_ct0 Entity
ENTITY dsp_d4_k32_ct0 IS
GENERIC
(
    CT0_D_C   : NATURAL := 4;
    CT0_K_C   : NATURAL := 32;
```

```
    CT0_ST_C  : NATURAL := 15
);
PORT
(
    -- Control io
    clk       : IN  STD_LOGIC;
    rst       : IN  STD_LOGIC;
    en        : IN  STD_LOGIC;
    valid     : OUT STD_LOGIC;
    -- Input Signals
    xv        : IN  CT0_X_ARRAY;
    a         : IN  CT0_A_TYPE;
    y         : IN  CT0_Y_TYPE;
    -- Output.
    ct0       : OUT CT0_TYPE
);
END dsp_d4_k32_ct0;
```

Listing A.8 provides the VHDL Entity for `dsp_d8_k16_ct0`. Classification Evaluation Pipeline. Listing A.9 provides the VHDL Entity for `dsp_d8_k32_ct0`. Classification Evaluation Pipeline.

***Listing A.8:*** *VHDL Entity:* `dsp_d8_k16_ct0`. *Classification Evaluation Pipeline.*

```
-- dsp_d8_k16_ct0 Entity
ENTITY dsp_d8_k16_ct0 IS
GENERIC
(
    CT0_D_C   : NATURAL := 8;
    CT0_K_C   : NATURAL := 16;
    CT0_ST_C  : NATURAL := 14
);
PORT
(
    -- Control io
    clk       : IN  STD_LOGIC;
    rst       : IN  STD_LOGIC;
    en        : IN  STD_LOGIC;
    valid     : OUT STD_LOGIC;
    -- Input Signals
    xv        : IN  CT0_X_ARRAY;
    a         : IN  CT0_A_TYPE;
    y         : IN  CT0_Y_TYPE;
    -- Output.
    ct0       : OUT CT0_TYPE
);
END dsp_d8_k16_ct0;
```

***Listing A.9:*** *VHDL Entity:* `dsp_d8_k32_ct0`. *Classification Evaluation Pipeline.*

```
-- dsp_d8_k32_ct0 Entity
ENTITY dsp_d8_k32_ct0 IS
GENERIC
(
    CT0_D_C   : NATURAL := 8;
    CT0_K_C   : NATURAL := 32;
    CT0_ST_C  : NATURAL := 15
);
PORT
(
```

```
    -- Control io
    clk        : IN  STD_LOGIC;
    rst        : IN  STD_LOGIC;
    en         : IN  STD_LOGIC;
    valid      : OUT STD_LOGIC;
    -- Input Signals
    xv         : IN  CT0_X_ARRAY;
    a          : IN  CT0_A_TYPE;
    y          : IN  CT0_Y_TYPE;
    -- Output.
    ct0        : OUT CT0_TYPE
);
END dsp_d8_k32_ct0;
```

Listing A.10 provides the VHDL Entity for `dsp_d2_k4_ce0`. Classification Evaluation Pipeline. Listing A.11 provides the VHDL Entity for `dsp_d2_k8_ce0`. Classification Evaluation Pipeline. Listing A.12 provides the VHDL Entity for `dsp_d2_k16_ce0`. Classification Evaluation Pipeline. Listing A.13 provides the VHDL Entity for `dsp_d2_k32_ce0`. Classification Evaluation Pipeline.

***Listing A.10:*** *VHDL Entity:* `dsp_d2_k4_ce0`. *Classification Evaluation Pipeline.*

```
-- dsp_d2_k4_ce0 Entity
ENTITY dsp_d2_k4_ce0 IS
GENERIC
(
    CE0_D_C    : NATURAL := 2;
    CE0_K_C    : NATURAL := 4;
    CE0_ST_C   : NATURAL := 8
);
PORT
(
    -- Ports go here
    clk        : IN  STD_LOGIC;
    rst        : IN  STD_LOGIC;
    en         : IN  STD_LOGIC;
    valid      : OUT STD_LOGIC;
    -- Input Signals
    xv         : IN  CE0_X_ARRAY;
    x          : IN  CE0_X_TYPE;
    xp         : IN  CE0_X_TYPE;
    xn         : IN  CE0_X_TYPE;
    a          : IN  CE0_A_TYPE;
    y          : IN  CE0_Y_TYPE;
    -- Output.
    ce0        : OUT CE0_TYPE
);
END dsp_d2_k4_ce0;
```

***Listing A.11:*** *VHDL Entity:* `dsp_d2_k8_ce0`. *Classification Evaluation Pipeline.*

```
-- dsp_d2_k8_ce0 Entity
ENTITY dsp_d2_k8_ce0 IS
GENERIC
(
    CE0_D_C    : NATURAL := 2;
    CE0_K_C    : NATURAL := 8;
    CE0_ST_C   : NATURAL := 9
);
```

```
PORT
(
    -- Ports go here
    clk        : IN  STD_LOGIC;
    rst        : IN  STD_LOGIC;
    en         : IN  STD_LOGIC;
    valid      : OUT STD_LOGIC;
    -- Input Signals
    xv         : IN  CE0_X_ARRAY;
    x          : IN  CE0_X_TYPE;
    xp         : IN  CE0_X_TYPE;
    xn         : IN  CE0_X_TYPE;
    a          : IN  CE0_A_TYPE;
    y          : IN  CE0_Y_TYPE;
    -- Output.
    ce0        : OUT CE0_TYPE
);
END dsp_d2_k8_ce0;
```

***Listing A.12:*** *VHDL Entity:* `dsp_d2_k16_ce0`*. Classification Evaluation Pipeline.*

```
-- dsp_d2_k16_ce0 Entity
ENTITY dsp_d2_k16_ce0 IS
GENERIC
(
    CE0_D_C    : NATURAL := 2;
    CE0_K_C    : NATURAL := 16;
    CE0_ST_C   : NATURAL := 10
);
PORT
(
    -- Ports go here
    clk        : IN  STD_LOGIC;
    rst        : IN  STD_LOGIC;
    en         : IN  STD_LOGIC;
    valid      : OUT STD_LOGIC;
    -- Input Signals
    xv         : IN  CE0_X_ARRAY;
    x          : IN  CE0_X_TYPE;
    xp         : IN  CE0_X_TYPE;
    xn         : IN  CE0_X_TYPE;
    a          : IN  CE0_A_TYPE;
    y          : IN  CE0_Y_TYPE;
    -- Output
    ce0        : OUT CE0_TYPE
);
END dsp_d2_k16_ce0;
```

***Listing A.13:*** *VHDL Entity:* `dsp_d2_k32_ce0`*. Classification Evaluation Pipeline.*

```
-- dsp_d2_k32_ce0 Entity
ENTITY dsp_d2_k32_ce0 IS
GENERIC
(
    CE0_D_C    : NATURAL := 2;
    CE0_K_C    : NATURAL := 32;
    CE0_ST_C   : NATURAL := 11
);
PORT
(
```

```
    -- Ports go here
    clk        : IN  STD_LOGIC;
    rst        : IN  STD_LOGIC;
    en         : IN  STD_LOGIC;
    valid      : OUT STD_LOGIC;
    -- Input Signals
    xv         : IN  CE0_X_ARRAY;
    x          : IN  CE0_X_TYPE;
    xp         : IN  CE0_X_TYPE;
    xn         : IN  CE0_X_TYPE;
    a          : IN  CE0_A_TYPE;
    y          : IN  CE0_Y_TYPE;
    -- Output.
    ce0        : OUT CE0_TYPE
);
END dsp_d2_k32_ce0;
```

Listing A.14 provides the VHDL Entity for `dsp_d4_k8_ce0`. Classification Evaluation Pipeline. Listing A.15 provides the VHDL Entity for `dsp_d4_k16_ce0`. Classification Evaluation Pipeline. Listing A.16 provides the VHDL Entity for `dsp_d4_k32_ce0`. Classification Evaluation Pipeline.

***Listing A.14:*** *VHDL Entity:* `dsp_d4_k8_ce0`. *Classification Evaluation Pipeline.*

```
-- dsp_d4_k8_ce0 Entity
ENTITY dsp_d4_k8_ce0 IS
GENERIC
(
    CE0_D_C    : NATURAL := 4;
    CE0_K_C    : NATURAL := 8;
    CE0_ST_C   : NATURAL := 9
);
PORT
(
    -- Ports go here
    clk        : IN  STD_LOGIC;
    rst        : IN  STD_LOGIC;
    en         : IN  STD_LOGIC;
    valid      : OUT STD_LOGIC;
    -- Input Signals
    xv         : IN  CE0_X_ARRAY;
    x          : IN  CE0_X_TYPE;
    xp         : IN  CE0_X_TYPE;
    xn         : IN  CE0_X_TYPE;
    a          : IN  CE0_A_TYPE;
    y          : IN  CE0_Y_TYPE;
    -- Output.
    ce0        : OUT CE0_TYPE
);
END dsp_d4_k8_ce0;
```

***Listing A.15:*** *VHDL Entity:* `dsp_d4_k16_ce0`. *Classification Evaluation Pipeline.*

```
-- dsp_d4_k16_ce0 Entity
ENTITY dsp_d4_k16_ce0 IS
GENERIC
(
    CE0_D_C    : NATURAL := 4;
    CE0_K_C    : NATURAL := 16;
```

```
    CE0_ST_C  : NATURAL := 10
);
PORT
(
    -- Ports go here
    clk       : IN  STD_LOGIC;
    rst       : IN  STD_LOGIC;
    en        : IN  STD_LOGIC;
    valid     : OUT STD_LOGIC;
    -- Input Signals
    xv        : IN  CE0_X_ARRAY;
    x         : IN  CE0_X_TYPE;
    xp        : IN  CE0_X_TYPE;
    xn        : IN  CE0_X_TYPE;
    a         : IN  CE0_A_TYPE;
    y         : IN  CE0_Y_TYPE;
    -- Output.
    ce0       : OUT CE0_TYPE
);
END dsp_d4_k16_ce0;
```

***Listing A.16:*** *VHDL Entity:* `dsp_d4_k32_ce0.` *Classification Evaluation Pipeline.*

```
-- dsp_d4_k32_ce0 Entity
ENTITY dsp_d4_k32_ce0 IS
GENERIC
(
    CE0_D_C   : NATURAL := 4;
    CE0_K_C   : NATURAL := 32;
    CE0_ST_C  : NATURAL := 11
);
PORT
(
    -- Ports go here
    clk       : IN  STD_LOGIC;
    rst       : IN  STD_LOGIC;
    en        : IN  STD_LOGIC;
    valid     : OUT STD_LOGIC;
    -- Input Signals
    xv        : IN  CE0_X_ARRAY;
    x         : IN  CE0_X_TYPE;
    xp        : IN  CE0_X_TYPE;
    xn        : IN  CE0_X_TYPE;
    a         : IN  CE0_A_TYPE;
    y         : IN  CE0_Y_TYPE;
    -- Output.
    ce0       : OUT CE0_TYPE
);
END dsp_d4_k32_ce0;
```

Listing A.17 provides the VHDL Entity for `dsp_d8_k16_ce0.` Classification Evaluation Pipeline. Listing A.18 provides the VHDL Entity for `dsp_d8_k32_ce0.` Classification Evaluation Pipeline.

***Listing A.17:*** *VHDL Entity:* `dsp_d8_k16_ce0.` *Classification Evaluation Pipeline.*

```
-- dsp_d8_k16_ce0 Entity
ENTITY dsp_d8_k16_ce0 IS
GENERIC
(
```

```
    CE0_D_C   : NATURAL := 8;
    CE0_K_C   : NATURAL := 16;
    CE0_ST_C  : NATURAL := 11
);
PORT
(
    -- Ports go here
    clk       : IN  STD_LOGIC;
    rst       : IN  STD_LOGIC;
    en        : IN  STD_LOGIC;
    valid     : OUT STD_LOGIC;
    -- Input Signals
    xv        : IN  CE0_X_ARRAY;
    x         : IN  CE0_X_TYPE;
    xp        : IN  CE0_X_TYPE;
    xn        : IN  CE0_X_TYPE;
    a         : IN  CE0_A_TYPE;
    y         : IN  CE0_Y_TYPE;
    -- Output.
    ce0       : OUT CE0_TYPE
);
END dsp_d8_k16_ce0;
```

***Listing A.18:*** *VHDL Entity:* `dsp_d8_k32_ce0`*. Classification Evaluation Pipeline.*

```
-- dsp_d8_k32_ce0 Entity
ENTITY dsp_d8_k32_ce0 IS
GENERIC
(
    CE0_D_C   : NATURAL := 8;
    CE0_K_C   : NATURAL := 32;
    CE0_ST_C  : NATURAL := 12
);
PORT
(
    -- Ports go here
    clk       : IN  STD_LOGIC;
    rst       : IN  STD_LOGIC;
    en        : IN  STD_LOGIC;
    valid     : OUT STD_LOGIC;
    -- Input Signals
    xv        : IN  CE0_X_ARRAY;
    x         : IN  CE0_X_TYPE;
    xp        : IN  CE0_X_TYPE;
    xn        : IN  CE0_X_TYPE;
    a         : IN  CE0_A_TYPE;
    y         : IN  CE0_Y_TYPE;
    -- Output.
    ce0       : OUT CE0_TYPE
);
END dsp_d8_k32_ce0;
```

Listing A.19 provides the VHDL Entity for `dsp_d2_k4_rt0`. Classification Evaluation Pipeline. Listing A.20 provides the VHDL Entity for `dsp_d2_k8_rt0`. Classification Evaluation Pipeline. Listing A.21 provides the VHDL Entity for `dsp_d2_k16_rt0`. Classification Evaluation Pipeline. Listing A.22 provides the VHDL Entity for `dsp_d2_k32_rt0`. Classification Evaluation Pipeline.

***Listing A.19:*** *VHDL Entity:* `dsp_d2_k4_rt0`. *Classification Evaluation Pipeline.*

```
-- dsp_d2_k4_rt0 Entity
ENTITY dsp_d2_k4_rt0 IS
GENERIC
(
    RT0_D_C   : NATURAL := 2;
    RT0_K_C   : NATURAL := 4;
    RT0_ST_C  : NATURAL := 10
);
PORT
(
    -- Control io
    clk       : IN  STD_LOGIC;
    rst       : IN  STD_LOGIC;
    en        : IN  STD_LOGIC;
    valid     : OUT STD_LOGIC;
    -- Input Signals
    xv        : IN  RT0_X_ARRAY;
    ad        : IN  RT0_A_TYPE;
    ac        : IN  RT0_A_TYPE;
    y         : IN  RT0_Y_TYPE;
    -- Output
    rt0       : OUT RT0_TYPE
);
END dsp_d2_k4_rt0;
```

***Listing A.20:*** *VHDL Entity:* `dsp_d2_k8_rt0`. *Classification Evaluation Pipeline.*

```
-- dsp_d2_k8_rt0 Entity
ENTITY dsp_d2_k8_rt0 IS
GENERIC
(
    RT0_D_C   : NATURAL := 2;
    RT0_K_C   : NATURAL := 8;
    RT0_ST_C  : NATURAL := 12
);
PORT
(
    -- Control io
    clk       : IN  STD_LOGIC;
    rst       : IN  STD_LOGIC;
    en        : IN  STD_LOGIC;
    valid     : OUT STD_LOGIC;
    -- Input Signals
    xv        : IN  RT0_X_ARRAY;
    ad        : IN  RT0_A_TYPE;
    ac        : IN  RT0_A_TYPE;
    y         : IN  RT0_Y_TYPE;
    -- Output
    rt0       : OUT RT0_TYPE
);
END dsp_d2_k8_rt0;
```

***Listing A.21:*** *VHDL Entity:* `dsp_d2_k16_rt0`. *Classification Evaluation Pipeline.*

```
-- dsp_d2_k16_rt0 Entity
ENTITY dsp_d2_k16_rt0 IS
GENERIC
(
```

```
    RT0_D_C   : NATURAL := 2;
    RT0_K_C   : NATURAL := 16;
    RT0_ST_C  : NATURAL := 14
);
PORT
(
    -- Control io
    clk       : IN  STD_LOGIC;
    rst       : IN  STD_LOGIC;
    en        : IN  STD_LOGIC;
    valid     : OUT STD_LOGIC;
    -- Input Signals
    xv        : IN  RT0_X_ARRAY;
    ad        : IN  RT0_A_TYPE;
    ac        : IN  RT0_A_TYPE;
    y         : IN  RT0_Y_TYPE;
    -- Output
    rt0       : OUT RT0_TYPE
);
END dsp_d2_k16_rt0;
```

***Listing A.22:*** *VHDL Entity:* `dsp_d2_k32_rt0`*. Classification Evaluation Pipeline.*

```
-- dsp_d2_k32_rt0 Entity
ENTITY dsp_d2_k32_rt0 IS
GENERIC
(
    RT0_D_C   : NATURAL := 2;
    RT0_K_C   : NATURAL := 32;
    RT0_ST_C  : NATURAL := 16
);
PORT
(
    -- Control io
    clk       : IN  STD_LOGIC;
    rst       : IN  STD_LOGIC;
    en        : IN  STD_LOGIC;
    valid     : OUT STD_LOGIC;
    -- Input Signals
    xv        : IN  RT0_X_ARRAY;
    ad        : IN  RT0_A_TYPE;
    ac        : IN  RT0_A_TYPE;
    y         : IN  RT0_Y_TYPE;
    -- Output.
    rt0       : OUT RT0_TYPE
);
END dsp_d2_k32_rt0;
```

Listing A.23 provides the VHDL Entity for `dsp_d4_k8_rt0`. Classification Evaluation Pipeline. Listing A.24 provides the VHDL Entity for `dsp_d4_k16_rt0`. Classification Evaluation Pipeline. Listing A.25 provides the VHDL Entity for `dsp_d4_k32_rt0`. Classification Evaluation Pipeline.

***Listing A.23:*** *VHDL Entity:* `dsp_d4_k8_rt0`*. Classification Evaluation Pipeline.*

```
-- dsp_d4_k8_rt0 Entity
ENTITY dsp_d4_k8_rt0 IS
GENERIC
(
```

```
    RT0_D_C   : NATURAL := 4;
    RT0_K_C   : NATURAL := 8;
    RT0_ST_C  : NATURAL := 12
);
PORT
(
    -- Control io
    clk       : IN  STD_LOGIC;
    rst       : IN  STD_LOGIC;
    en        : IN  STD_LOGIC;
    valid     : OUT STD_LOGIC;
    -- Input Signals
    xv        : IN  RT0_X_ARRAY;
    ad        : IN  RT0_A_TYPE;
    ac        : IN  RT0_A_TYPE;
    y         : IN  RT0_Y_TYPE;
    -- Output
    rt0       : OUT RT0_TYPE
);
END dsp_d4_k8_rt0;
```

***Listing A.24:*** *VHDL Entity:* `dsp_d4_k16_rt0`*. Classification Evaluation Pipeline.*

```
-- dsp_d4_k16_rt0 Entity
ENTITY dsp_d4_k16_rt0 IS
GENERIC
(
    RT0_D_C   : NATURAL := 4;
    RT0_K_C   : NATURAL := 16;
    RT0_ST_C  : NATURAL := 14
);
PORT
(
    -- Control io
    clk       : IN  STD_LOGIC;
    rst       : IN  STD_LOGIC;
    en        : IN  STD_LOGIC;
    valid     : OUT STD_LOGIC;
    -- Input Signals
    xv        : IN  RT0_X_ARRAY;
    ad        : IN  RT0_A_TYPE;
    ac        : IN  RT0_A_TYPE;
    y         : IN  RT0_Y_TYPE;
    -- Output
    rt0       : OUT RT0_TYPE
);
END dsp_d4_k16_rt0;
```

***Listing A.25:*** *VHDL Entity:* `dsp_d4_k32_rt0`*. Classification Evaluation Pipeline.*

```
-- dsp_d4_k32_rt0 Entity
ENTITY dsp_d4_k32_rt0 IS
GENERIC
(
    RT0_D_C   : NATURAL := 4;
    RT0_K_C   : NATURAL := 32;
    RT0_ST_C  : NATURAL := 16
);
PORT
(
```

```
    -- Control io
    clk        : IN  STD_LOGIC;
    rst        : IN  STD_LOGIC;
    en         : IN  STD_LOGIC;
    valid      : OUT STD_LOGIC;
    -- Input Signals
    xv         : IN  RT0_X_ARRAY;
    ad         : IN  RT0_A_TYPE;
    ac         : IN  RT0_A_TYPE;
    y          : IN  RT0_Y_TYPE;
    -- Output
    rt0        : OUT RT0_TYPE
);
END dsp_d4_k32_rt0;
```

Listing A.26 provides the VHDL Entity for `dsp_d8_k16_rt0`. Classification Evaluation Pipeline. Listing A.27 provides the VHDL Entity for `dsp_d8_k32_rt0`. Classification Evaluation Pipeline.

***Listing A.26:*** *VHDL Entity:* `dsp_d8_k16_rt0`. *Classification Evaluation Pipeline.*

```
-- dsp_d8_k16_rt0 Entity
ENTITY dsp_d8_k16_rt0 IS
GENERIC
(
    RT0_D_C    : NATURAL := 8;
    RT0_K_C    : NATURAL := 16;
    RT0_ST_C   : NATURAL := 15
);
PORT
(
    -- Control io
    clk        : IN  STD_LOGIC;
    rst        : IN  STD_LOGIC;
    en         : IN  STD_LOGIC;
    valid      : OUT STD_LOGIC;
    -- Input Signals
    xv         : IN  RT0_X_ARRAY;
    ad         : IN  RT0_A_TYPE;
    ac         : IN  RT0_A_TYPE;
    y          : IN  RT0_Y_TYPE;
    -- Output
    rt0        : OUT RT0_TYPE
);
END dsp_d8_k16_rt0;
```

***Listing A.27:*** *VHDL Entity:* `dsp_d8_k32_rt0`. *Classification Evaluation Pipeline.*

```
-- dsp_d8_k32_rt0 Entity
ENTITY dsp_d8_k32_rt0 IS
GENERIC
(
    RT0_D_C    : NATURAL := 8;
    RT0_K_C    : NATURAL := 32;
    RT0_ST_C   : NATURAL := 17
);
PORT
(
    -- Control io
    clk        : IN  STD_LOGIC;
```

```
    rst        : IN  STD_LOGIC;
    en         : IN  STD_LOGIC;
    valid      : OUT STD_LOGIC;
    -- Input Signals
    xv         : IN  RT0_X_ARRAY;
    ad         : IN  RT0_A_TYPE;
    ac         : IN  RT0_A_TYPE;
    y          : IN  RT0_Y_TYPE;
    -- Output
    rt0        : OUT RT0_TYPE
);
END dsp_d8_k32_rt0;
```

Listing A.28 provides the VHDL Entity for `dsp_d2_k4_re0`. Classification Evaluation Pipeline. Listing A.29 provides the VHDL Entity for `dsp_d2_k8_re0`. Classification Evaluation Pipeline. Listing A.30 provides the VHDL Entity for `dsp_d2_k16_re0`. Classification Evaluation Pipeline. Listing A.31 provides the VHDL Entity for `dsp_d2_k32_re0`. Classification Evaluation Pipeline.

***Listing A.28:*** *VHDL Entity:* `dsp_d2_k4_re0`. *Classification Evaluation Pipeline.*

```
-- dsp_d2_k4_re0 Entity
ENTITY dsp_d2_k4_re0 IS
GENERIC
(
    RE0_D_C    : NATURAL := 2;
    RE0_K_C    : NATURAL := 4;
    RE0_ST_C   : NATURAL := 10
);
PORT
(
    -- Control io
    clk        : IN  STD_LOGIC;
    rst        : IN  STD_LOGIC;
    en         : IN  STD_LOGIC;
    valid      : OUT STD_LOGIC;
    -- Input Signals
    xv         : IN  RE0_X_ARRAY;
    x          : IN  RE0_X_TYPE;
    ad         : IN  RE0_A_TYPE;
    ac         : IN  RE0_A_TYPE;
    y          : IN  RE0_Y_TYPE;
    -- Output.
    re0        : OUT RE0_TYPE
);
END dsp_d2_k4_re0;
```

***Listing A.29:*** *VHDL Entity:* `dsp_d2_k8_re0`. *Classification Evaluation Pipeline.*

```
-- dsp_d2_k8_re0 Entity
ENTITY dsp_d2_k8_re0 IS
GENERIC
(
    RE0_D_C    : NATURAL := 2;
    RE0_K_C    : NATURAL := 8;
    RE0_ST_C   : NATURAL := 12
);
PORT
(
```

```
    -- Control io
    clk        : IN  STD_LOGIC;
    rst        : IN  STD_LOGIC;
    en         : IN  STD_LOGIC;
    valid      : OUT STD_LOGIC;
    -- Input Signals
    xv         : IN  RE0_X_ARRAY;
    x          : IN  RE0_X_TYPE;
    ad         : IN  RE0_A_TYPE;
    ac         : IN  RE0_A_TYPE;
    y          : IN  RE0_Y_TYPE;
    -- Output.
    re0        : OUT RE0_TYPE
);
END dsp_d2_k8_re0;
```

***Listing A.30:*** *VHDL Entity:* `dsp_d2_k16_re0`*. Classification Evaluation Pipeline.*

```
-- dsp_d2_k16_re0 Entity
ENTITY dsp_d2_k16_re0 IS
GENERIC
(
    RE0_D_C    : NATURAL := 2;
    RE0_K_C    : NATURAL := 16;
    RE0_ST_C   : NATURAL := 14
);
PORT
(
    -- Control io
    clk        : IN  STD_LOGIC;
    rst        : IN  STD_LOGIC;
    en         : IN  STD_LOGIC;
    valid      : OUT STD_LOGIC;
    -- Input Signals
    xv         : IN  RE0_X_ARRAY;
    x          : IN  RE0_X_TYPE;
    ad         : IN  RE0_A_TYPE;
    ac         : IN  RE0_A_TYPE;
    y          : IN  RE0_Y_TYPE;
    -- Output.
    re0        : OUT RE0_TYPE
);
END dsp_d2_k16_re0;
```

***Listing A.31:*** *VHDL Entity:* `dsp_d2_k32_re0`*. Classification Evaluation Pipeline.*

```
-- dsp_d2_k32_re0 Entity
ENTITY dsp_d2_k32_re0 IS
GENERIC
(
    RE0_D_C    : NATURAL := 2;
    RE0_K_C    : NATURAL := 32;
    RE0_ST_C   : NATURAL := 16
);
PORT
(
    -- Control io
    clk        : IN  STD_LOGIC;
    rst        : IN  STD_LOGIC;
    en         : IN  STD_LOGIC;
```

```
    valid     : OUT STD_LOGIC;
    -- Input Signals
    xv        : IN  RE0_X_ARRAY;
    x         : IN  RE0_X_TYPE;
    ad        : IN  RE0_A_TYPE;
    ac        : IN  RE0_A_TYPE;
    y         : IN  RE0_Y_TYPE;
    -- Output
    re0       : OUT RE0_TYPE
);
END dsp_d2_k32_re0;
```

Listing A.32 provides the VHDL Entity for `dsp_d4_k8_re0`. Classification Evaluation Pipeline. Listing A.33 provides the VHDL Entity for `dsp_d4_k16_re0`. Classification Evaluation Pipeline. Listing A.34 provides the VHDL Entity for `dsp_d4_k32_re0`. Classification Evaluation Pipeline.

***Listing A.32:*** *VHDL Entity:* `dsp_d4_k8_re0`. *Classification Evaluation Pipeline.*

```
-- dsp_d4_k8_re0 Entity
ENTITY dsp_d4_k8_re0 IS
GENERIC
(
    RE0_D_C   : NATURAL := 4;
    RE0_K_C   : NATURAL := 8;
    RE0_ST_C  : NATURAL := 12
);
PORT
(
    -- Control io
    clk       : IN  STD_LOGIC;
    rst       : IN  STD_LOGIC;
    en        : IN  STD_LOGIC;
    valid     : OUT STD_LOGIC;
    -- Input Signals
    xv        : IN  RE0_X_ARRAY;
    x         : IN  RE0_X_TYPE;
    ad        : IN  RE0_A_TYPE;
    ac        : IN  RE0_A_TYPE;
    y         : IN  RE0_Y_TYPE;
    -- Output.
    re0       : OUT RE0_TYPE
);
END dsp_d4_k8_re0;
```

***Listing A.33:*** *VHDL Entity:* `dsp_d4_k16_re0`. *Classification Evaluation Pipeline.*

```
-- dsp_d4_k16_re0 Entity
ENTITY dsp_d4_k16_re0 IS
GENERIC
(
    RE0_D_C   : NATURAL := 4;
    RE0_K_C   : NATURAL := 16;
    RE0_ST_C  : NATURAL := 14
);
PORT
(
    -- Control io
    clk       : IN  STD_LOGIC;
```

```
    rst        : IN  STD_LOGIC;
    en         : IN  STD_LOGIC;
    valid      : OUT STD_LOGIC;
    -- Input Signals
    xv         : IN  RE0_X_ARRAY;
    x          : IN  RE0_X_TYPE;
    ad         : IN  RE0_A_TYPE;
    ac         : IN  RE0_A_TYPE;
    y          : IN  RE0_Y_TYPE;
    -- Output.
    re0        : OUT RE0_TYPE
);
END dsp_d4_k16_re0;
```

***Listing A.34:*** *VHDL Entity:* `dsp_d4_k32_re0`*. Classification Evaluation Pipeline.*

```
-- dsp_d4_k32_re0 Entity
ENTITY dsp_d4_k32_re0 IS
GENERIC
(
    RE0_D_C    : NATURAL := 4;
    RE0_K_C    : NATURAL := 32;
    RE0_ST_C   : NATURAL := 16
);
PORT
(
    -- Control io
    clk        : IN  STD_LOGIC;
    rst        : IN  STD_LOGIC;
    en         : IN  STD_LOGIC;
    valid      : OUT STD_LOGIC;
    -- Input Signals
    xv         : IN  RE0_X_ARRAY;
    x          : IN  RE0_X_TYPE;
    ad         : IN  RE0_A_TYPE;
    ac         : IN  RE0_A_TYPE;
    y          : IN  RE0_Y_TYPE;
    -- Output.
    re0        : OUT RE0_TYPE
);
END dsp_d4_k32_re0;
```

Listing A.35 provides the VHDL Entity for `dsp_d8_k16_re0`. Classification Evaluation Pipeline. Listing A.36 provides the VHDL Entity for `dsp_d8_k32_re0`. Classification Evaluation Pipeline.

***Listing A.35:*** *VHDL Entity:* `dsp_d8_k16_re0`*. Classification Evaluation Pipeline.*

```
-- dsp_d8_k16_re0 Entity
ENTITY dsp_d8_k16_re0 IS
GENERIC
(
    RE0_D_C    : NATURAL := 8;
    RE0_K_C    : NATURAL := 16;
    RE0_ST_C   : NATURAL := 15
);
PORT
(
    -- Control io
    clk        : IN  STD_LOGIC;
```

```
    rst        : IN  STD_LOGIC;
    en         : IN  STD_LOGIC;
    valid      : OUT STD_LOGIC;
    -- Input Signals
    xv         : IN  RE0_X_ARRAY;
    x          : IN  RE0_X_TYPE;
    ad         : IN  RE0_A_TYPE;
    ac         : IN  RE0_A_TYPE;
    y          : IN  RE0_Y_TYPE;
    -- Output.
    re0        : OUT RE0_TYPE
);
END dsp_d8_k16_re0;
```

***Listing A.36:*** *VHDL Entity:* `dsp_d8_k32_re0`*. Classification Evaluation Pipeline.*

```
-- dsp_d8_k32_re0 Entity
ENTITY dsp_d8_k32_re0 IS
GENERIC
(
    RE0_D_C    : NATURAL := 8;
    RE0_K_C    : NATURAL := 32;
    RE0_ST_C   : NATURAL := 17
);
PORT
(
    -- Control io
    clk        : IN  STD_LOGIC;
    rst        : IN  STD_LOGIC;
    en         : IN  STD_LOGIC;
    valid      : OUT STD_LOGIC;
    -- Input Signals
    xv         : IN  RE0_X_ARRAY;
    x          : IN  RE0_X_TYPE;
    ad         : IN  RE0_A_TYPE;
    ac         : IN  RE0_A_TYPE;
    y          : IN  RE0_Y_TYPE;
    -- Output.
    re0        : OUT RE0_TYPE
);
END dsp_d8_k32_re0;
```

# References


[1] M. Al Rabieah and C. Bouganis, "Fpga based nonlinear support vector machine training using an ensemble learning," in *Proc. of the 25th International Conference on Field Programmable Logic and Applications (FPL)*, 09 2015, pp. 1–4.

[2] S. Afifi, H. GholamHosseini, and R. Sinha, *Hardware Acceleration of SVM-Based Classifier for Melanoma Images*, F. Huang and A. Sugimoto, Eds. Cham: Springer International Publishing, 2016. [Online]. Available: https://doi.org/10.1007/978-3-319-30285-0_19

[3] S. Afifi, H. Gholamhosseini, and R. Sinha, "A low-cost fpga-based svm classifier for melanoma detection," in *IEEE-EMBS Conference on Biomedical Engineering and Sciences*, L. Khuan, Ed. Kuala Lumpur, Malaysia, Dec 2016.

[4] B. E. Boser, I. Guyon, and V. Vapnik, "A training algorithm for optimal margin classifiers," in *Proc. of the 5th Annual Workshop on Computational Learning Theory*, Pittsburgh, PA, July 1992, pp. 144–152.

[5] F. Rosenblatt, "The perceptron: A probabilistic model for information storage and organization in the brain," *Psychological Review*, vol. 65, no. 6, pp. 386–408, 1958.

[6] G. E. Moore, "Cramming more components onto integrated circuits," *Electronics Magazine*, vol. 38, no. 8, pp. 114–117, April 1965.

[7] R. Kurzweil, *The Age of Intelligent Machines*. Cambridge, MA: The MIT Press, 1990.

[8] R. Kurzweil, *The Age of Spiritual Machines: When Computers Exceed Human Intelligence*, 1st ed. New York, NY: Penguin USA, 1999.

[9] R. Kurzweil, *The Singularity Is Near: When Humans Transcend Biology*, 1st ed. New York, NY: Viking, 2005.

[10] J. S. Kilby, "Miniaturised electronic circuits," U.S. Patent 3 138 743, June 23, 1964.

[11] R. N. Noyce, "Semiconductor device-and-lead structure," U.S. Patent 2 981 877, April 25, 1961.

[12] (2012) IEEE Global History Network Milestones: First semiconductor integrated circuit (IC), 1958. [Online]. Available: http://ethw.org/Milestones:First_Semiconductor_Integrated_Circuit_(IC),_1958

[13] S. Haykin, *Neural Networks: A Comprehensive Foundation*, 2nd ed. Upper Saddle River, NJ: Prentice-Hall, Inc., 1998.

[14] M. Minsky and S. Papert, *Perceptrons: An Introduction to Computational Geometry*. Cambridge, MA: The MIT Press, 1969.

[15] H. P. Newquist, *The Brain Makers*, 1st ed. Santa Barbara, CA: Editors and Engineers, Limited, 1994.

[16] I. Stewart, *Does God Play Dice? The New Mathematics of Chaos*, 2nd ed. Malden, MA: Blackwell Publishing, 2002.

[17] J. Gleick, *Chaos: Making a New Science*, 2nd ed. New York, NY: Penguin Books, 2008.

[18] E. N. Lorenz, "Deterministic nonperiodic flow," *Journal of Atmospheric Sciences*, vol. 20, no. 2, pp. 130–148, 1963.

[19] B. B. Mandelbrot, *The Fractal Geometry of Nature*, 1st ed. New York, NY: Henry Holt and Company, 1982.

[20] D. Hebb, *The Organization of Behavior*. New York, NY: Wiley & Sons, 1949.

[21] T. D. Sanger, "Optimal unsupervised learning in a single-layer linear feedforward neural network," *Neural Netw.*, vol. 2, pp. 459–473, 1989.

[22] J. J. Hopfield and D. W. Tank, "Neural computation of decisions in optimization problems," *Biological Cybernetics*, vol. 52, no. 3, pp. 141–152, July 1985.

[23] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning internal representations by error propagation," in *Parallel Distributed Processing: Explorations in the Microstructure of Cognition; Foundations*, D. E. Rumelhart, J. L. McClelland, and the PDP Research Group, Eds. Cambridge, MA: The MIT Press, 1986, vol. 1.

[24] R. H. Freeman, "Configurable electrical circuit having configurable logic elements and configurable interconnects," U.S. Patent 4 870 302, September 26, 1989.

[25] C. Cortes and V. N. Vapnik, "Support-vector networks," *Machine Learning*, vol. 20, pp. 273–297, September 1995.

[26] V. N. Vapnik, *The Nature of Statistical Learning Theory*, 1st ed. New York, NY: Springer-Verlag, 1995.

[27] V. N. Vapnik, *Statistical Learning Theory*, 1st ed. New York, NY: John Wiley & Sons, Inc., 1998.

[28] V. N. Vapnik, "An overview of statistical learning theory," *IEEE Trans. Neural Netw.*, vol. 10, no. 5, pp. 988–999, September 1999.

[29] R. K. Rajkumar, "Pipeline defect prediction using support vector machine," Ph.D. dissertation, Dept. of Elect. and Electron. Eng., Univ. of Nottingham, Kuala Lumpur, Malaysia, 2011.

[30] D. Poole, *Linear Algebra: A Modern Introduction*, 2nd ed. Belmont, CA: Thomson Brooks/Cole, 2006.

[31] L. Hamel, *Knowledge Discovery with Support Vector Machines*, 1st ed. New York, NY: John Wiley & Sons, Inc., 2009.

[32] R. K. Sundaram, *A First Course in Optimization Theory*, 1st ed. Cambridge, UK: Cambridge University Press, 1996.

[33] D. S. G. Pollock, *A Handbook of Time Series Analysis, Signal Processing and Dynamics*, 1st ed. San Diego, CA: Academic Press, 1999.

[34] K. Ogata, *Modern Control Engineering*, 4th ed. Upper Saddle River, NJ: Prentice-Hall, Inc., 2002.

[35] R. H. Shumway and D. S. Stoffer, *Time Series Analysis and Its Applications*, 3rd ed. New York, NY: Springer, 2011.

[36] D. G. Manolakis, V. K. Ingle, and S. M. Kogon, *Statistical and Adaptive Signal Processing: Spectral Estimation, Signal Modelling, Adaptive Filtering, and Array Processing*, 1st ed. Norwood, MA: Artech House, Inc., 2005.

[37] A. Cichocki and R. Unbehauen, *Neural Networks for Optimization and Signal Processing*, 1st ed. New York, NY: John Wiley & Sons, Inc., 1993.

[38] V. N. Vapnik and A. Y. Chervonenkis, "On the uniform convergence of relative frequencies of events to their probabilities," *Theory of Probability and its Applications*, vol. 16, no. 2, pp. 264–280, 1971.

[39] V. N. Vapnik, *Estimation of Dependences Based on Empirical Data: Springer Series in Statistics*, 1st ed. Secaucus, NJ: Springer-Verlag New York, Inc., 1982.

[40] J. C. Platt, "Fast training of support vector machines using sequential minimal optimization," in *Advances in Kernel Methods*, B. Schölkopf, C. J. C. Burges, and A. J. Smola, Eds. Cambridge, MA: The MIT Press, 1999, pp. 185–208.

[41] E. Osuna, R. Freund, and F. Girosi, "An improved training algorithm for support vector machines," in *Proceedings of the 1997 IEEE Workshop on Neural Networks for Signal Processing*, Amelia Island, FL, September 1997, pp. 276–285.

[42] L. S. Lasdon, A. D. Waren, A. Jain, and M. Ratner, "Design and testing of a generalized reduced gradient code for nonlinear programming," *ACM Transactions on Mathematical Software (TOMS)*, vol. 4, no. 1, pp. 34–50, March 1978.

[43] S. Smith and L. S. Lasdon, "Solving large sparse nonlinear programs using GRG," *INFORMS Journal on Computing*, vol. 4, no. 1, pp. 2–15, February 1992.

[44] M. Jaggi, "Revisiting Frank-Wolfe: Projection-free sparse convex optimization," in *Proceedings of the 30th International Conference on Machine Learning (ICML-13)*, S. Dasgupta and D. Mcallester, Eds., vol. 28, no. 1, Atlanta, GA, June 2013, pp. 427–435.

[45] E. G. Gilbert, "An iterative procedure for computing the minimum of a quadratic form on a convex set," *SIAM J. Control*, vol. 4, no. 1, pp. 61–80, 1966.

[46] L. Chang, H. Qiao, A. Wan, and J. Keane, "An improved Gilbert algorithm with rapid convergence," in *Proc. of the 2006 IEEE/RSJ Int. Conf. on Intelligent Robots and Systems*, Beijing, China, October 2006, pp. 3861–3866.

[47] J. A. Hartigan, *Clustering Algorithms*, 1st ed. New York, NY: John Wiley & Sons, Inc., 1975.

[48] J. A. Hartigan and M. A. Wong, "Algorithm AS 136: A k-means clustering algorithm," *Journal of the Royal Statistical Society. Series C (Applied Statistics)*, vol. 28, no. 1, pp. 100–108, 1979.

[49] D. Anguita, A. Boni, and S. Ridella, "SVM learning with fixed-point math," in *Proc. of the IEEE Int. Joint Conf. on Neural Netw. (IJCNN)*, Portland, OR, July 2003, pp. 2072–2076.

[50] D. Anguita, A. Boni, and S. Ridella, "A digital architecture for support vector machines: Theory, algorithm, and FPGA implementation," *IEEE Trans. Neural Netw.*, vol. 14, no. 5, pp. 993–1009, September 2003.

[51] S. Kim, S. Lee, K. Min, and K. Cho, "Design of support vector machine circuit for real-time classification," in *Proc. of the IEEE Int. Symp. on Integrated Circuits, 2011 (ISIC 2011)*, Singapore, Singapore, December 2011, pp. 384–387.

[52] D. Mahmoodi, A. Soleimani, H. Khosravi, and M. Taghizadeh, "FPGA simulation of linear and nonlinear support vector machine," *Journal of Software Engineering and Applications*, vol. 5, no. 4, pp. 320–328, 2011.

[53] K. Irick, M. DeBole, V. Narayanan, and A. Gayasen, "A hardware efficient support vector machine architecture for FPGA," in *Proc. of the 16th IEEE Int. Symp. on Field-Programmable Custom Computing Machines, 2008 (FCCM 2008)*, Palo Alto, CA, April 2008, pp. 304–305.

[54] M. Ruiz-Llata, G. Guarnizo, and M. Yébenes-Calvino, "FPGA implementation of a support vector machine for classification and regression," in *Proc. of the IEEE Int. Joint Conf. on Neural Netw. (IJCNN)*, Barcelona, Spain, July 2010, pp. 477–481.

[55] J. G. Filho, M. Raffo, M. Strum, and W. J. Chau, "A general-purpose dynamically reconfigurable SVM," in *Proc. of the IEEE VI Southern Programmable Logic Conf. (SPL)*, Ipojuca, Brazil, March 2010, pp. 107–112.

[56] R. A. Patil, G. Gupta, V. Sahula, and A. S. Mandal, "Power aware hardware prototyping of multiclass SVM classifier through reconfiguration," in *Proc. of the IEEE 2012 25th Int. Conf. on VLSI Design (VLSID)*, Hyderabad, India, January 2012, pp. 62–67.

[57] M. E. Mavroforakis, M. Sdralis, and S. Theodoridis, "A novel SVM geometric algorithm based on reduced convex hulls," in *Proc. of the 18th IEEE Int. Conf. on Pattern Recognition (ICPR)*, Hong Kong, China, August 2006, pp. 564–568.

[58] S. Martin, "Training support vector machines using Gilbert's algorithm," in *Proc. of the 5th IEEE Int. Conf. on Data Mining (ICDM)*, Houston, TX, November 2005, pp. 306–313.

[59] M. Papadonikolakis and C. Bouganis, "Efficient FPGA mapping of Gilbert's algorithm for SVM training on large-scale classification problems," in *Proc. of the IEEE Int. Conf. on Field Programmable Logic and Applications (FPL)*, Heidelberg, Germany, September 2008, pp. 385–390.

[60] M. Papadonikolakis and C. Bouganis, "A scalable FPGA architecture for non-linear SVM training," in *Proc. of the IEEE Int. Conf. on Field-Programmable Technology (FPT)*, Taipei, Taiwan, December 2008, pp. 337–340.

[61] M. Papadonikolakis, C. Bouganis, and G. Constantinides, "Performance comparison of GPU and FPGA architectures for the SVM training problem," in *Proc. of the IEEE Int. Conf. on Field-Programmable Technology (FPT)*, Sydney, Australia, December 2009, pp. 388–391.

[62] Z. B. Liu, J. G. Liu, and Z. Chen, "A generalized Gilbert's algorithm for approximating general SVM classifiers," *Neurocomputing*, vol. 73, no. 1-3, pp. 219–224, December 2009.

[63] M. Papadonikolakis and C. Bouganis, "A heterogeneous FPGA architecture for support vector machine training," in *Proc. of the IEEE Symp. on Field-Programmable Custom Computing Machines*, Charlotte, NC, May 2010, pp. 211–214.

[64] M. Papadonikolakis and C. Bouganis, "A novel FPGA-based SVM classifier," in *Proc. of IEEE Int. Conf. on Field-Programmable Technology (FPT)*, Beijing, China, December 2010, pp. 283–286.

[65] M. Papadonikolakis and C. Bouganis, "Novel cascade FPGA accelerator for support vector machines classification," *IEEE Trans. Neural Netw. and Learning Systems*, vol. 23, no. 75, pp. 1040–1052, July 2012.

[66] C. Maxfield, *The Design Warrior's Guide to FPGAs*, 1st ed. Orlando, FL: Academic Press, Inc., 2004.

[67] S. Brown and Z. Vranesic, *Fundamentals of Digital Logic with VHDL Design*, 3rd ed. New York, NY: McGraw-Hill, Inc., 2008.

[68] R. Woods, J. Mcallister, R. Turner, Y. Yi, and G. Lightbody, *FPGA-based Implementation of Signal Processing Systems*, 1st ed. Chichester, UK: John Wiley & Sons, Inc., 2008.

[69] "Accelerating DSP designs with the total 28-nm DSP portfolio," White Paper, Altera Corporation, April 2011.

[70] H. T. Kung and C. E. Leiserson, "Systolic arrays (for VLSI)," in *SIAM Sparse Matrix Proceedings 1978*, I. S. Duff and G. W. Stewart, Eds. Philadelphia, PA: SIAM Press, 1979, pp. 256–282.

[71] *IEEE ASSP Magazine, VLSI Array Processors*, vol. 2, no. 3, July 1985.

[72] S. Y. Kung, *VLSI Array Processors*, 1st ed. Englewood Cliffs, NJ: Prentice Hall, 1988.

[73] S. Y. Kung, "VLSI array processors: Designs and applications," in *Proc. of the IEEE Int. Symp. on Circuits and Systems, 1988*, vol. 1, Espoo, Finland, June 1988, pp. 313–320.

[74] G. M. Amdahl, "Validity of the single processor approach to achieving large scale computing capabilities," in *Proc. of the American Federation of Information Processing Societies (AFIPS) Spring Joint Computer Conf.*, vol. 30. Atlantic City, N.J.: AFIPS Press, Reston, Va., April 1967, pp. 483–485.

[75] G. Goslin, "Using xilinx FPGAs to design custom digital signal processing devices," in *Proc. of the 1995 DSPX Technical Program Conference and Exhibition*, San Jose, CA, January 1995, pp. 595–604.

[76] J. C. Sprott, *Chaos and Time-series Analysis*, 1st ed. Oxford, UK: Oxford University Press, 2003.

[77] T. Kapitaniak and S. R. Bishop, *The Illustrated Dictionary of Nonlinear Dynamics and Chaos*, 1st ed. New York, NY: John Wiley & Sons, Inc., 1999.

[78] L. Glass and M. C. Mackey, *From Clocks to Chaos: The Rhythms of Life*, 1st ed. Princeton, NJ: Princeton University Press, 1988.

[79] D. J. Albers and J. C. Sprott, "Routes to chaos in neural networks with random weights," *International Journal of Bifurcation and Chaos*, vol. 8, no. 7, pp. 1463–1478, May 1998.

[80] F. Takens, "Detecting strange attractors in turbulence," in *Dynamical Systems and Turbulence*, ser. Lecture Notes in Mathematics, D. A. Rand and L. S. Young, Eds. Berlin, Germany: Springer-Verlag, 1981, vol. 898, pp. 366–381.

[81] H. D. I. Abarbanel, *Analysis of Observed Chaotic Data*, 1st ed. New York, NY: Springer-Verlag, 1996.

[82] C. E. Shannon and W. Weaver, *The Mathematical Theory of Communication*, 2nd ed. Champaign, IL: University of Illinios Press, 1963.

[83] A. M. Fraser and H. L. Swinney, "Independent coordinates for strange attractors from mutual information," *Physical Review A*, vol. 33, no. 2, pp. 1134–1140, February 1986.

[84] A. M. Fraser, "Information and entropy in strange attractors," *IEEE Trans. on Inf. Theory*, vol. 35, no. 2, pp. 245–262, March 1989.

[85] M. B. Kennel, R. Brown, and H. D. I. Abarbanel, "Determining embedding dimension for phase-space reconstruction using a geometrical construction," *Physical Review A*, vol. 45, no. 6, pp. 3403–3411, March 1992.

[86] P. B. A. Phear, R. K. Rajkumar, and D. Isa, "Efficient non-iterative fixed-period SVM training architecture for FPGAs," in *Proc. of the 39th Annu. Conf. of the IEEE Industrial Electronics Society (IECON 2013)*, Vienna, Austria, November 2013.

[87] (2016) Altera Quartus Prime. Altera Corporation. [Online]. Available: https://www.altera.com/products/design-software/fpga-design/quartus-prime/overview.highResolutionDisplay.html

[88] (2016) Vivado Design Suite. Xilinx Inc. [Online]. Available: http://www.xilinx.com/products/design-tools/vivado.html

[89] (2016) ModelSim-Altera Edition Software. Mentor Graphics. [Online]. Available: https://www.altera.com/products/design-software/model---simulation/modelsim-altera-software.highResolutionDisplay.html

[90] (2016) MathWorks MATLAB. The MathWorks, Inc. [Online]. Available: https://au.mathworks.com/products/matlab/

[91] (2016) GNU Octave. Free Software Foundation, Inc. [Online]. Available: https://gnu.org/software/octave/

[92] H. Kantz and T. Schreiber, *Nonlinear Time Series Analysis*, 2nd ed. New York, NY: Cambridge University Press, 2003.

[93] (2016) GCC, the GNU Compiler Collection. Free Software Foundation, Inc. [Online]. Available: https://gcc.gnu.org/

[94] J. Vinet and A. Griffin. (2016) Arch Linux. [Online]. Available: https://www.archlinux.org/

[95] (2016) Git. Software Freedom Conservancy, Inc. [Online]. Available: https://git-scm.com/

[96] (2016) Bitbucket - the Git solution for professional teams. Atlassian. [Online]. Available: https://bitbucket.org/

[97] (2016) LaTeX - A document preparation system. LaTeX Project. [Online]. Available: https://www.latex-project.org/

[98] T. Williams and C. Kelley. (2016) gnuplot homepage. [Online]. Available: http://www.gnuplot.info/

[99] O. Cheong. (2016) The Ipe extensible drawing editor. [Online]. Available: http://ipe.otfried.org/

[100] R. R. Curtin, J. R. Cline, N. P. Slagle, W. B. March, P. Ram, N. A. Mehta, and A. G. Gray, "MLPACK: A scalable C++ machine learning library," *Journal of Machine Learning Research*, vol. 14, pp. 801–805, 2013.

[101] C.-C. Chang and C.-J. Lin, "LIBSVM: A library for support vector machines," *ACM Transactions on Intelligent Systems and Technology*, vol. 2, no. 3, pp. 27:1–27:27, 2011, software available at http://www.csie.ntu.edu.tw/~cjlin/libsvm.

[102] *Stratix V Device Datasheet*, 5SGSMD5, Altera Corporation, December 2015.

[103] *Stratix V Device Handbook*, 5SGSMD5, Altera Corporation, June 2016.

[104] K. Kleine. (2015, September) perf: Linux profiling with performance counters. Linux Kernel Organization, Inc. [Online]. Available: https://perf.wiki.kernel.org/index.php/Main_Page

[105] T. M. Thompson, *From Error-Correcting Codes Through Sphere Packings to Simple Groups*, 1st ed., ser. The Carus Mathematical Monographs. Washington, DC: Mathematical Association of America, 1983, vol. 21.