THE UNIVERSITY OF SYDNEY

DOCTORAL THESIS

Algorithms and Architectures for High Performance Kernel Adaptive Filtering

Author:
Nicholas FRASER

Supervisor:
Prof. Philip LEONG
Associate Supervisor:
Assoc. Prof. Craig JIN

A thesis submitted in fulfillment of the requirements for the degree of Doctor of Philosophy

in the

Computer and Software Engineering Laboratory
School of Electrical and Information Engineering

September 14, 2020

Declaration of Authorship

I, Nicholas FRASER, declare that this thesis titled, "Algorithms and Architectures for High Performance Kernel Adaptive Filtering" and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a research degree at this University.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.
- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

Signed:

Date:

Abstract

In recent years, machine learning algorithms have been taking over traditional programming approaches and are being used to great effect in a broad range of applications. In this thesis, we look at kernel adaptive filters (KAFs): a class of non-linear adaptive filters which utilise a Mercer kernel function. These algorithms have been shown to provide high accuracy in a number of applications, including those that would require extremely high data rates, such as channel equalisation, extremely low latency, such as financial market prediction, and many other applications from embedded to cloud computing.

The purpose of this thesis is to determine whether or not it is feasible to apply KAFs to such a range of applications. To do this, we propose new KAF algorithms which are hardware friendly in nature. We also explore exotic computing platforms from field programmable gate arrays (FPGAs) to distributed computing settings. We show that with these techniques, orders of magnitude increases in performance can be achieved and as such, KAFs can be applied to a wide range of problems.

Acknowledgements

I would like to thank my supervisor, Professor Philip Leong and my co-supervisor Associate Professor Craig Jin for their support and guidance throughout my PhD. I'd like to thank my former and current colleagues including: Michael Frechtling, Duncan Moss, Stephen Tridgell, Calla Klafas, Nicolas Epain, Andrew Wabnitz, Abhaya Parthy, Aengus Martin, Joseph Prinable, Sepideh Noohi, Michaela Blott, Yaman Umuroğlu, Giulio Gambardella, Ken O'Brien, Johannes de Fine Licht, Lisa Halder, Brian Colgan, Alessandro Pappalardo, Lucian Petrica, Thomas Preußer, Kees Vissers and Peter Ogden.

Thank you also to my friends and family for being supportive and understanding throughout the duration of my PhD. In particular, my parents: Pat and Ken Fraser. You have always provided incredible support and advice. If I have become even half as good a person as either of you, I can feel confident I will contribute positively to the world. My brothers: Tom and Mitch Fraser. You have always been a source of inspiration, motivation and support. To my friends (there are too many to thank - apologies if I left you out): Joel Cotter, David Smith, Gareth Bryant, Andrew Phelan, Bethany Green, Owen Brasier, Melanie Roddy, Bassam Barake, Zachary Dickman, Peter McColgan, Eloi Marcel de Bazelaire, Julio Ruiz, Ana Ateca, Caoímhe Louise Cassidy, Marta Zerolo, Scewit Ghebreweld, Laura Clesceri, Monica Epifano, Joan Farjo, Sheena Hillier, Mathilde Poutrel, Andrea Tomljenović, Iva Sutlović, Marianne Zankoski, Orlagh Lavelle and Melissa Dowd, thank you for your friendship and support.

Authorship Attribution Statement

Several chapters of this thesis are based on publications made in the course of my PhD candidature at the Computer Engineering Laboratory (CELAB), the Department of Electrical and Information Engineering, the University of Sydney, under the supervision of Professor Philip Leong. The specific contributions made to these publications is outlined below:

Chapter 4 is published as Fraser et al. (2017a), which is in turn a journal extension of Fraser et al. (2015a). The original idea behind this publication is from Philip Leong and me. The core was designed and implemented by JunKyu Lee, Duncan Moss and me. The folding architecture was designed and implemented by me. The exponential evaluation module was designed and implemented by Stephen Tridgell. The system implementation was designed and implemented by Duncan Moss and Julian Faraone. The random search was designed by Philip Leong and was implemented by Julian Faraone. The central processing unit (CPU) and graphics processing unit (GPU) implementations were done by me. The manuscripts were written by Junkyu Lee, Julian Faraone, Duncan Moss, Philip Leong and me.

Chapter 5 is published as Fraser and Leong (2020). The original idea, design, implementation, and all experiments were by me. The idea of using Remez for exponential evaluation was from Philip Leong. The manuscript was written by Philip Leong and me.

Chapter 6 is published as Fraser et al. (2015b). The original idea was from Philip Leong and Duncan Moss. The theoretical analysis was by Nicolas Epain and me. Initial experiments were conducted by Duncan Moss. All experimental results in the paper were conducted by me. The manuscript was written by Nicolas Epain, Duncan Moss, Philip Leong and me.

Appendix A is published as Fraser et al. (2017b). The original idea was from Yaman Umuroğlu. The core implementation was by Yaman Umuroğlu, Giulio Gambardella and me. The neural network training experiments were by me. The system implementation was by Yaman Umuroğlu and Giulio Gambardella. The manuscript was written by Yaman Umuroğlu, Giulio Gambardella, Michaela Blott and me.

In addition to the statements above, in cases where I am not the corresponding author of a published item, permission to include the published material has been granted by the corresponding author.

Nicholas James Fraser,

As supervisor for the candidature upon which this thesis is based, I can confirm that the authorship attribution statements above are correct.

Professor Philip Leong,

Statement of Originality

This is to certify that to the best of my knowledge, the content of this thesis is my own work. This thesis has not been submitted for any degree or other purposes.

I certify that the intellectual content of this thesis is the product of my own work and that all the assistance received in preparing this thesis and sources have been acknowledged.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation and Aims

Machine learning has been defined generally as a “Field of study that gives computers the ability to learn without being explicitly programmed” (Simon, 2013). In recent years it has grown in popularity as it has been found to have almost limitless applications. Often the algorithms associated with machine learning are surprisingly simple but allow computers to perform a variety of complex tasks. These algorithms also provide solutions to several computational problems for which no explicit solutions have been found including:

- computer vision, such as image classification (Krizhevsky, Sutskever, and Hinton, 2012), object detection (Redmon and Farhadi, 2017) and semantic segmentation (He et al., 2017);
- recommender systems (Naumov et al., 2019); and
- natural language processing, such as speech recognition (Amodei et al., 2016).

Difficult problems which are solved with machine learning implementations often suffer from high computational complexity and latency. Many non-trivial machine learning problems require large amounts of training examples with many features in order to perform adequately. Furthermore, in online learning problems training examples are not known beforehand. The machine learning algorithm must be able to provide predictions while continuously learning in order to improve future predictions. Online tracking algorithms must also be able to provide accurate predictions even if there are changes in the underlying unknown process.

Online tracking problems often require real-time learning and predictions in order to be useful. These applications include:

- channel equalisation (Engel, Mannor, and Meir, 2004);
- anomaly detection in networks (Moustafa and Slay, 2015);
- prediction of financial data (Silva et al., 2016); and
- audio signal processing, such as dereverberation (Naylor and Gaubitch, 2005) and audio compression (Silva et al., 2016).

A recent group of algorithms, known as KAFs (Liu, Príncipe, and Haykin, 2011) have shown promise in addressing such problems. However, little work has been done to demonstrate the *feasibility* of utilising such algorithms in the above applications domains, but rather most prior works have simply shown that there are accuracy benefits for using such algorithms. Compared with linear adaptive filters, KAFs exhibit a moderate increase in algorithmic and computational complexity, along with increased storage requirements (Liu, Príncipe, and Haykin, 2011).

In this thesis, we attempt to bridge this gap. Specifically, we ask ourselves the following research question: "can algorithmic enhancements and custom accelerators show the feasibility of applying KAFs to a wide range of application domains?" Throughout this thesis, we show that KAFs may be applicable in a wide range of areas from learning small scale online models with extreme data rate requirements (Chapter 5), to learning extremely large models on a distributed computing platform (Chapter 6). We show that with careful algorithmic selection and custom computing architectures, orders of magnitude in overall performance improvements are possible.

1.2 Contributions

This thesis presents two methods which can be applied to KAFs in order to parallelise them in two distinct ways, through pipelining and through data parallelism.

Firstly, we introduce delayed model adaptation, which previously had only been applied to linear adaptive filters (Long, Ling, and Proakis, 1989), which allows KAFs to be pipelined in a way which overcomes some dependency issues. We demonstrate its effectiveness by modifying the kernel normalised least mean squares (KNLMS) (Richard, Bermudez, and Honeine, 2009) to create a family of similar algorithms. Specifically, the delayed kernel normalised least mean squares (DKNLMS), multi-delayed kernel normalised least mean squares (MDKNLMS), delayed kernel normalised least mean squares with dictionary guarding (DKNLMS-DG) and delayed kernel normalised least mean squares with correction terms (DKNLMS-CT) algorithms. Secondly, we demonstrate a method to parallelise kernel recursive least squares (KRLS) in a way that significantly reduces the interdependencies between compute nodes, doesn't require multiple iterations through the dataset, but still provides an upper bound on the overall modelling error. The resultant algorithm, distributed kernel recursive least squares (DistKRLS), is amenable to acceleration on distributed platforms.

We propose scalable architectures and subsequent implementations of core generators capable of implementing: KNLMS, DKNLMS, MDKNLMS and DKNLMS-DG in Chisel (Bachrach et al., 2012). We also provide a library containing highly performant C and CUDA implementations for standard CPUs / GPUs respectively. The CPU library significantly outperforms prior works (Van Vaerenbergh, 2012) for smaller scale models, while the GPU library is the only implementation of KAFs of which we're aware. Both of these libraries and the Chisel-based core generators have been open sourced.[1][2] We also provide a Vivado HLS-based (Xilinx, 2016) architecture and implementation for scalable, foldable, deeply pipelined implementations of KNLMS which are suitable for hyperparameter optimisation, or for online predictive modelling of multi-channel systems.

We perform analysis of various hardware platforms and algorithmic analysis of various KAFs to understand their amenability to highly performant hardware implementations. Finally, we perform dynamic rounding analysis of various KAFs using Monte Carlo arithmetic (MCA) (Parker, 1997).

[1] https://bitbucket.org/nick_fraser/libkaf
[2] https://github.com/nickfraser/rosetta

1.3 Structure of the Thesis

This thesis is organised as follows:

- Chapter 2 provides a background in machine learning, computer arithmetic and we review some prior computer architectures with a focus on online kernel based algorithms known as KAFs;
- Chapter 3 identifies and compares some available computing platforms and studies properties (in particular, arithmetic intensity (AI) and roundoff error analysis) of some popular KAFs to determine which computing platforms would be best suited as deployment targets under various conditions;
- Chapter 4 considers the problem of hyperparameter optimisation (or multi-channel online learning problems) for smaller scale KAFs and proposes a core generator, along with several FPGA implementations capable of extremely high throughput;
- Chapter 5 considers extremely high throughput applications, such as channel equalisation, and proposes a new class of KAFs along with a core generator capable of generating FPGA implementation capable of handling extremely high data rates;
- Chapter 6 looks at decomposing large scale, offline KAF problems into smaller scale problems capable of being distributed and later recombined while maintaining high accuracy and avoiding multiple iterations over the training set;
- Conclusions of this work are discussed in Chapter 7 along with potential future work;
- Lastly, Chapter A contains the work of a paper which formed part of my PhD work, but does not fit into the main story of this thesis on the subject of FPGA implementations of BNNs.

In terms of the structure of this thesis, Chapter 3, Chapter 4, Chapter 5 and Chapter 6 consider vastly different application domains. As such, much of the introductory and background material associated with each chapter is retained within the chapter itself. The background chapter of this thesis then provides general background required for the reader to quickly pickup the background material associated with the content chapter. In terms of content chapters, they relate to the following publications which were published during the course of my PhD studies:

- Chapter 3 is related to no specific publication, but can be seen as core material which guides the decision making process for the subsequent chapters;
- Chapter 4 is heavily based on Fraser et al. (2015a) and Fraser et al. (2017a), Fraser et al. (2017a) being a journal extension of Fraser et al. (2015a);
- Chapter 5 is heavily based on Fraser and Leong (2020);
- Chapter 6 is based on Fraser et al. (2015b) while also containing significant extra material;
- Finally A is heavily based on Fraser et al. (2017b).

Chapter 2

Background

This section provides a background in machine learning, kernel methods, error analysis in finite precision arithmetic and an overview of FPGA architectures for machine learning.

2.1 Machine Learning

Machine learning refers to a field of study in which computers are taught to perform tasks without being *explicitly* programmed to do so (Samuel, 1959; Koza et al., 1996). In recent years, with the increase in computational resources available, machine learning has been found to be a simple and effective tool in solving many complex algorithm design problems. Such problems include:

a) series forecasting (Tay and Cao, 2001);

b) channel equalisation (Engel, Mannor, and Meir, 2004);

c) image classification (Krizhevsky, Sutskever, and Hinton, 2012); and

d) object detection (Redmon and Farhadi, 2017).

In this work, we focus on non-linear adaptive filtering tasks, such as time-series prediction and channel equalisation. Adaptive filters are well known online algorithms which can model linear systems and have modest computational requirements. Extending them to address non-linear problems raises a number of challenges which will be discussed in this section. A relatively new set of algorithms, known as KAFs (Liu, Príncipe, and Haykin, 2011), address several of these issues by introducing the *kernel trick* (Cortes and Vapnik, 1995) and this forms a foundation for the work in this thesis. However, before we discuss the specifics of KAFs, we must understand linear regression, adaptive filters and the so-called kernel trick.

In this section, we aim to lay the foundation for several common machine learning concepts and KAFs. In subsequent chapters, we build on this foundation and describe in detail the specific KAF associated with each content chapter.

2.1.1 Linear Regression

Linear regression is commonly used in many machine learning problems and in statistics. Let $\mathbf{X} = \{\mathbf{x}_1, \mathbf{x}_2, \cdots, \mathbf{x}_N\} \in \mathbb{R}^{N\times M}$ be the input training set of N observations of dimension M, and the target be $\mathbf{y} \in \mathbb{R}^N$. Least-squares linear regression attempts to find the optimal vector $\mathbf{h} \in \mathbb{R}^{\mathbf{M}}$ which satisfies

$$J = \min \|\mathbf{y} - \mathbf{X}\mathbf{h}\|^2 \quad . \tag{2.1}$$

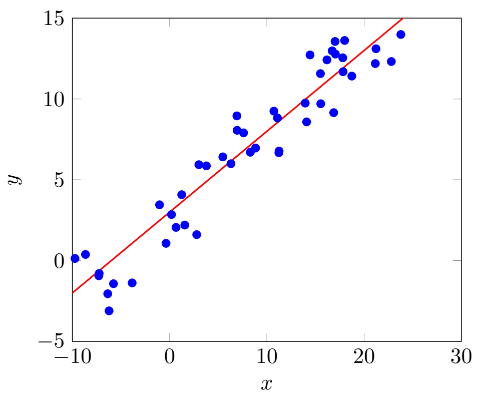


FIGURE 2.1: Example of linear regression applied to a 2D problem.

This is a minima when

$$\frac{\partial}{\partial \mathbf{h}}(\mathbf{y}-\mathbf{X}\mathbf{h})^T(\mathbf{y}-\mathbf{X}\mathbf{h}) = -2\mathbf{X}^T\mathbf{y} + 2\mathbf{X}^T\mathbf{X}\mathbf{h} = 0 \ . \tag{2.2}$$

Linear regression is often used to find a linear trend in some noisy data. Figure 2.1 shows linear regression applied to a 2-dimensional problem, where the entries of $\mathbf{h}$ represent the y-intercept and gradient of the line. Note that linear regression can be applied to higher order problems in which case the model is assumed to be a plane or hyper-plane. There are many iterative methods for finding the optimal solution for $\mathbf{h}$ including batch gradient descent (BGD) (Cauchy, 1847) algorithm shown in Algorithm 2.1. Note that $x_{i,j}$ refers to the j^{th} element of $\mathbf{x}_i$.

Algorithm 2.1 Pseudocode for the BGD algorithm. Note η is the step-size parameter and the hypothesis function, $f_h(x)$, is given by $\mathbf{h}^T\mathbf{X}$ for linear regression problems.

```
Initialise $\mathbf{h} = \mathbf{0}$. Choose a step size parameter $\eta$.
while Not converged do
    for $j = 1, 2, \cdots, m$ do
        $\mathbf{h}_j = \mathbf{h}_j - \eta\frac{1}{n}\sum_{i=1}^{n}(f_h(x_i) - \mathbf{y})x_{i,j}$
    end for
end while
```

BGD is a very common algorithm as it is simple to implement and can be used for a range of hypothesis functions, $f_h(x)$. Note that batch gradient descent algorithm slowly converges towards optimal values for the weights, h. Batch gradient descent can be quite slow in practice as it needs to iterate through all of the training data before making a single “step” towards an optimal solution for h. A modification to BGD known as mini batch gradient descent improves the convergence of BGD by breaking the training sets into k smaller groups of size N/k. Using the mini BGD algorithm, a single step can be taken after iterating through N/k training examples. For least squares linear regression problems if $(\mathbf{X}^T\mathbf{X})$ is not singular, a direct solution can be computed for $\mathbf{h}$ as follows:

$$\mathbf{h} = (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}\mathbf{y} \ . \tag{2.3}$$

Finally, if an approximate solution for $\mathbf{h}$ is sufficient, another common technique to find $\mathbf{h}$ is stocastic gradient descent (SGD) (Robbins and Monro, 1951). SGD has gained popularity in recent years as it is an optimiser commonly used in training on deep neural networks.

Linear regression assumes that the model which underlies the data is linear. Unfortunately, for non-trivial problems this is often not the case. Consider the four data sets, shown in Figure 2.2, which all produce the same linear regression model.[1] These datasets are known as Anscombe's quartet (Anscombe, 1973), In particular, note that in the set denoted by x_2 there is a relationship between input and output but it is a non-linear relationship which is not modelled well using a linear model. Anscombe

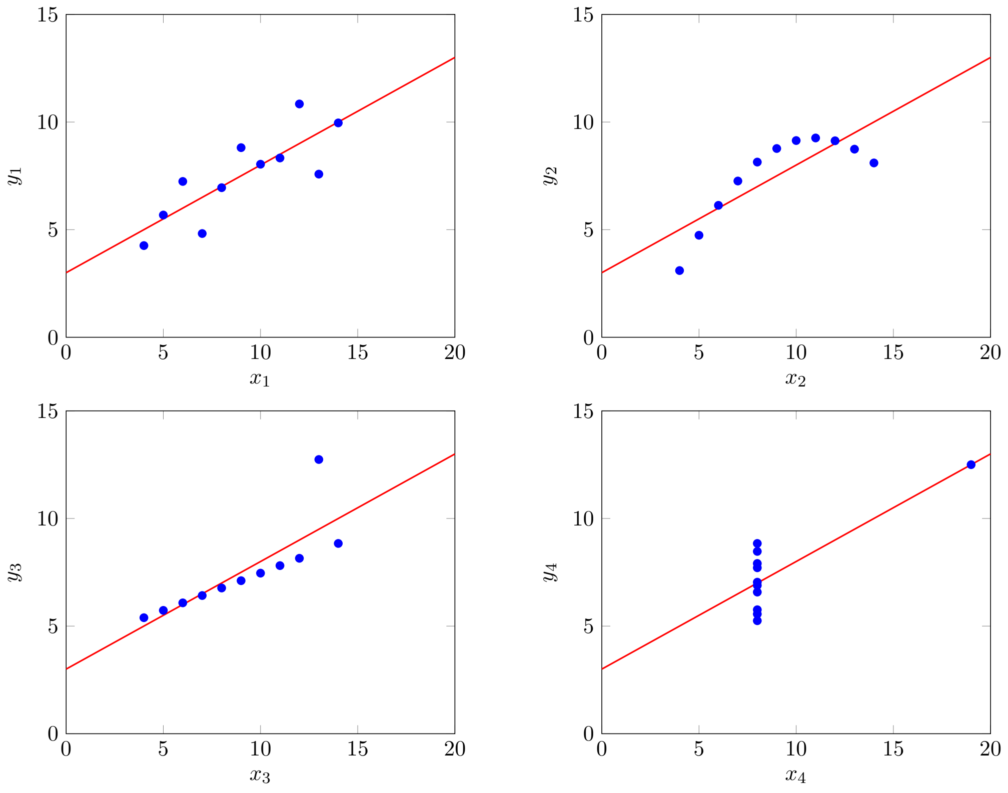


FIGURE 2.2: Anscombe's quartet - four completely different data sets with almost exactly the same statistical properties.

argues the need for plotting data, instead of relying on statistical measurements. For the four datasets in Figure 2.2 several statistical measurements of the data are almost exactly the same including: mean, variance, equation of regression lines, estimated standard error and several others (Anscombe, 1973). Although high dimensional spaces are difficult to graph in machine learning, it highlights the need to understand your dataset in order to avoid making modelling mistakes.

2.1.2 Non-linear Regression

A common way to adapt linear regression to non-linear problems is to apply a non-linear mapping, $\Phi(x) \in \mathbb{R}^M \rightarrow \mathbb{R}^{M'}$, to the input data. Linear regression can then be applied to this new data to find $\tilde{\mathbf{h}} \in \mathbb{R}^{M'}$. The cost function for this new space becomes:

$$J' = \min_{\tilde{\mathbf{h}}} \left\| \mathbf{y} - \tilde{\mathbf{X}}\tilde{\mathbf{h}} \right\|^2 \quad . \tag{2.4}$$

[1]Within a tolerance of 1%

where $\tilde{\mathbf{X}} = [\Phi(\mathbf{x}_1), \Phi(\mathbf{x}_2), \cdots, \Phi(\mathbf{x}_n)]^T \in \mathbb{R}^{N \times M'}$. For most non-linear mapping functions, $\Phi(x)$, an optimal solution for $\tilde{\mathbf{h}}$ can be found using BGD or the normal equation, i.e., Equation (2.3). Using the set denoted by x_2, we can create a mapping function, Φ_{x_2}, which allows us to model x_2 effectively. By looking at the shape created by the data in x_2, we may guess that there is a parabolic relationship between input and output. Given this guess, consider the following mapping function:

$$\Phi_{x_2}(\mathbf{x}) = \left[x_0, x_1, x_1^2\right]^T \quad , \tag{2.5}$$

where $x_0 = 1$ is the bias term and x_1 is the x-coordinate of the current input sample. Figure 2.3 shows Anscombe's quartet modelled using Φ_{x_2} as the mapping function. Notice that in particular, x_2 is modelled almost perfectly by this new mapping

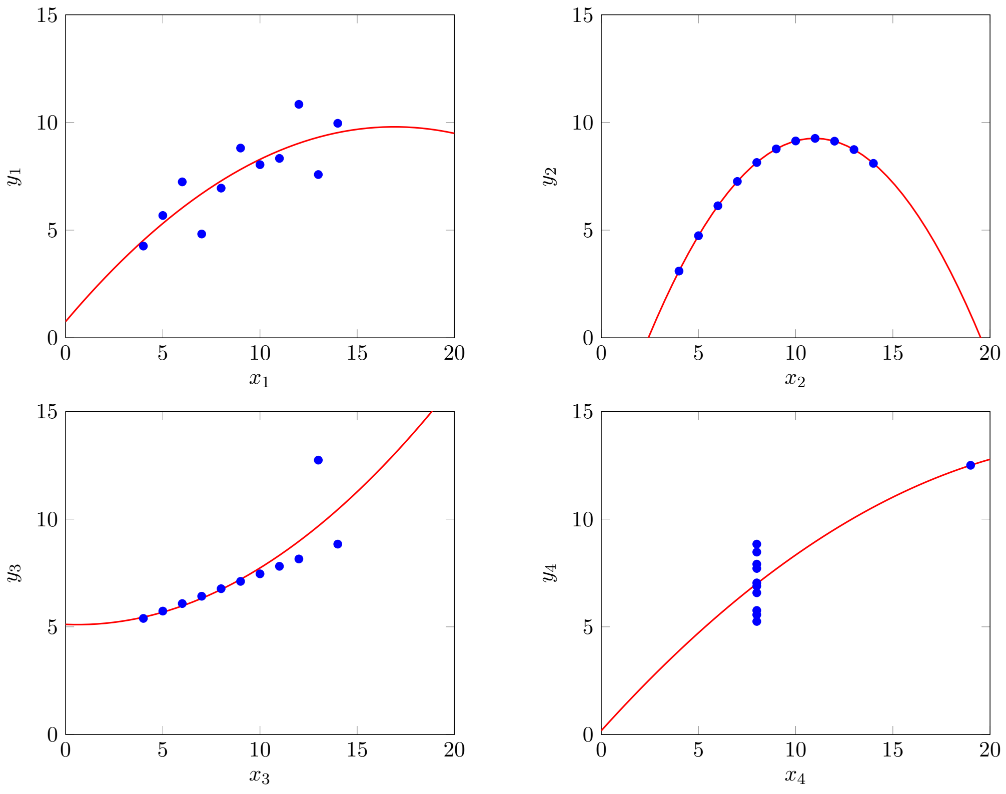


FIGURE 2.3: Anscombe's quartet - modelled using the mapping function Φ_{x_2}.

function. Similarly, other more complex systems can be modelled well by using linear regression and a non-linear mapping function. However, in the context of machine learning, there can be several challenges in taking this approach, in particular the selection of Φ.

In many instances, machine learning algorithms are used to model so-called "black box" systems in which the relationship between input and output is unknown or difficult to observe. For example, it's not clear how one would design feature mapping functions to model:

a) how a set of pixels map to decimal digits (LeCun et al., 1998);

b) how much electricity a city will consume at a particular time, based on the time of the year and the electricity consumed in the several days proceeding the current day (Silva et al., 2013); and

c) whether a particular currency will increase or decrease in value, based on prior values of several different currencies (Silva et al., 2016).

Particularly for high dimensional input data, designing such functions can involve sophisticated learning techniques, such as grammatical evolution (Silva et al., 2013), multi-variate Taylor Series (Taylor, 1715) or Volterra Series (Volterra, 1887) expansions. These techniques can incur significant computational costs themselves. For complex models, the dimension of M' can dramatically increase, causing a significant increase in computational complexity of regression algorithms. This increase in complexity is often referred to as the *curse of dimensionality* (Bellman, 1957).

2.1.3 Kernel Methods

Unfortunately, in many applications the feature space, M', is so large that a real time implementation would not be practical. To find an efficient solution which minimises J', the problem needs to be framed slightly differently. To do this, we need to represent our linear regression problem in the *dual representation*. The dual representation can be obtained by pre-multiplying Equation (2.3) by the identity $(\mathbf{X}^T\mathbf{X})(\mathbf{X}^T\mathbf{X})^{-1}$ to obtain:

$$\begin{aligned}\mathbf{h} &= (\mathbf{X}^T\mathbf{X})(\mathbf{X}^T\mathbf{X})^{-1}(\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\mathbf{y}\\ \mathbf{h} &= \mathbf{X}^T\boldsymbol{\alpha}\ \ ,\end{aligned} \tag{2.6}$$

which makes $\mathbf{h} = \sum_{i=1}^{N} \alpha_i \mathbf{x}_i$ a linear combination of the training set (Shawe-Taylor and Cristianini, 2004).

Let $\mathbf{K} = \tilde{\mathbf{X}}\tilde{\mathbf{X}}^T$ be the kernel matrix where $K_{i,j}$, the entry corresponding to the ith row and jth column of the kernel matrix, is given by the kernel function, $\kappa(\mathbf{x}_i, \mathbf{x}_j)$. For a standard linear regression problem the kernel function is the dot product between to input vectors $\kappa(\mathbf{x}_i, \mathbf{x}_j) = \mathbf{x}_i^T\mathbf{x}_j$. With this new representation, the cost function can be updated to the following:

$$J'' = \min_{\boldsymbol{\alpha}} \|\mathbf{y} - \mathbf{K}\boldsymbol{\alpha}\|^2 \ \ , \tag{2.7}$$

where $\boldsymbol{\alpha}$ is a $N \times 1$ vector of weights. Using Equation (2.2), the minima of J' can be found using the dual representation as follows:

$$\begin{aligned}\mathbf{X}^T\mathbf{X}\mathbf{h} &= \mathbf{X}^T\mathbf{y}\\ \mathbf{X}\mathbf{X}^T\mathbf{X}\mathbf{X}^T\boldsymbol{\alpha} &= \mathbf{X}\mathbf{X}^T\mathbf{y}\\ \mathbf{K}^2\boldsymbol{\alpha} &= \mathbf{K}\mathbf{y}\\ \boldsymbol{\alpha} &= \mathbf{K}^{-1}\mathbf{y}\ \ .\end{aligned} \tag{2.8}$$

To find the entries of the matrix $\mathbf{K}$, only the dot product between two mapped vectors needs to be calculated, $\kappa(\mathbf{x}_i, \mathbf{x}_j) = \Phi(\mathbf{x}_i)^T\Phi(\mathbf{x}_j)$. Kernel functions provide a solution for $\kappa(\mathbf{x}_i, \mathbf{x}_j)$ without directly mapping the inputs vectors. For example, consider the polynomial kernel:

$$\kappa(\mathbf{x}_i, \mathbf{x}_j) = (\mathbf{x}_i^T\mathbf{x}_j + c)^d \ \ , \tag{2.9}$$

where c is a constant bias and $d \in \mathbb{N}$ is the order of the polynomial. In order to calculate its corresponding mapping function, let us first consider a simple example. Let $\mathbf{x}, \mathbf{v} \in \mathbb{R}^M$, where $M = 2$ and $\kappa(\mathbf{x}, \mathbf{v}) = (\mathbf{x}^T\mathbf{v} + c)^d$, where $c = 1$ and $d = 2$.

Note, for this example we switch the notation from $\kappa(\mathbf{x}_i, \mathbf{x}_j) \to \kappa(\mathbf{x}, \mathbf{v})$ to improve readability of the subsequent indexing. We wish to find a mapping function $\Phi(\mathbf{x}) \in \mathbb{R}^2 \to \mathbb{R}^{M'}$, such that:

$$\kappa(\mathbf{x}, \mathbf{v}) = \Phi(\mathbf{x})^T \Phi(\mathbf{v}) = (\mathbf{x}^T \mathbf{v} + c)^d = (\mathbf{x}^T \mathbf{v} + 1)^2 \tag{2.10}$$
$$\kappa(\mathbf{x}, \mathbf{v}) = (x_1 v_1 + x_2 v_2 + 1)^2 \tag{2.11}$$
$$\kappa(\mathbf{x}, \mathbf{v}) = x_1^2 v_1^2 + x_2^2 v_2^2 + 2x_1 v_1 x_2 v_2 + 2x_1 v_1 + 2x_2 v_2 + 1 \ , \tag{2.12}$$

where x_i/v_i are the i^{th} terms of $\mathbf{x}/\mathbf{v}$ respectively. By grouping terms associated with $\mathbf{x}$ and $\mathbf{v}$, we can see that $\Phi(\mathbf{x}) = \left[x_1^2, x_2^2, \sqrt{2}x_1 x_2, \sqrt{2}x_1, \sqrt{2}x_2, 1\right]^T$.

Following a similar procedure as in the previous example, further generalisation of Equation (2.12) for when $d = 2$ and $M = m$ leads to the following mapping function:

$$\kappa(\mathbf{x}, \mathbf{v}) = \Phi(\mathbf{x})^T \Phi(\mathbf{v}) = (\mathbf{x}^T \mathbf{v} + 1)^2 \tag{2.13}$$
$$\kappa(\mathbf{x}, \mathbf{v}) = (x_1 v_1 + \cdots + x_m v_m + 1)^2 \tag{2.14}$$
$$\kappa(\mathbf{x}, \mathbf{v}) = \sum_{i=1}^{m} x_i^2 v_i^2 + \sum_{i=1}^{m} \sum_{j=1}^{m} x_i v_j + \sum_{i=1}^{m} x_i v_i + 1 \ . \tag{2.15}$$

Again, by carefully grouping terms, we end up with the following:

$$\Phi(\mathbf{x}) = \left[x_1^2, \cdots, x_m^2, \sqrt{2}x_m x_{m-1}, \cdots, \sqrt{2}x_m x_1, \cdots, \sqrt{2}x_2 x_1, \sqrt{2}x_m, \cdots, \sqrt{2}x_1, 1\right]^T \ . \tag{2.16}$$

The calculation of the mapping function for the second order polynomial kernel has a computational complexity of $O(m^2)$ while the equivalent kernel function is of order $O(m)$.

We now further generalise Equation (2.16) for arbitrary values of d. Without loss of generality, we set $c = 0$, and following similar steps to Shashua (2009, Chapter 4.3), we get:

$$\kappa(\mathbf{x}, \mathbf{v}) = \Phi(\mathbf{x})^T \Phi(\mathbf{v}) = (\mathbf{x}^T \mathbf{v})^d \tag{2.17}$$
$$\kappa(\mathbf{x}, \mathbf{v}) = (x_1 v_1 + \cdots + x_m v_m)^d \tag{2.18}$$
$$\kappa(\mathbf{x}, \mathbf{v}) = \sum_{\sum_{i=1}^{m} n_i = d} \binom{d}{n_1, \cdots, n_m} \prod_{i=1}^{m} x_i^{n_i} v_i^{n_i} \ , \tag{2.19}$$

where $n_i \in \mathbb{N} \ \forall \ i$ and $\binom{d}{n_1, \cdots, n_m}$ is the multinomial coefficient, given by:

$$\binom{d}{n_1, \cdots, n_m} = \frac{d!}{\prod_{i=1}^{m} n_i!} \ . \tag{2.20}$$

An interpretation of Equation (2.19) is that every term in the sum is a unique permutation of the multi-index $\mathbf{n} = (n_1, \cdots, n_m) \in \mathbb{N}_0^m$ such that $\sum_i^m n_i = d$, e.g., a unique arrangement of packing d balls into m bins. Using multi-index notation, we can write Equation (2.19) compactly as follows:

$$\kappa(\mathbf{x}, \mathbf{v}) = \sum_{|\mathbf{n}| = d} \binom{d}{\mathbf{n}} \mathbf{x}^{\mathbf{n}} \mathbf{v}^{\mathbf{n}} \ . \tag{2.21}$$

Following on from this, the polynomial mapping function can be written as follows:

$$\Phi(\mathbf{x}) = \left(\sqrt{\binom{d}{\mathbf{n}}}\mathbf{x}^{\mathbf{n}}\right)_{|\mathbf{n}|=d} \quad . \tag{2.22}$$

This formulation can be used for cases where $c \neq 0$, by simply augmenting the input vectors, i.e., $\hat{\mathbf{x}} = \left[\mathbf{x}^T, \sqrt{c}\right]^T$.

The number of terms, M', in $\Phi(\mathbf{x})$ is given by the number of possible combinations of the multi-index, $\mathbf{n}$. As stated previously, this is analogous to the number of unique packing of d balls into m bins, or more concisely as:

$$M' = \binom{m+d-1}{m-1} \tag{2.23}$$

$$M' = \binom{m+d-1}{d} \tag{2.24}$$

$$M' = \frac{(m+d-1)!}{d!(m-1)!} \tag{2.25}$$

In terms of scalability, the calculation of $\Phi(\mathbf{x})$ is of order $O(m^d)$ complexity, while the calculation of $\kappa(\mathbf{x}, \mathbf{v})$ is order $O(m+d)$ complexity.

One of the most common kernel functions (and the one which is the main focus in this work) is the Gaussian kernel, given by:

$$\kappa(\mathbf{x}, \mathbf{v}) = e^{-\frac{\|\mathbf{x}-\mathbf{v}\|^2}{2\sigma^2}} \quad , \tag{2.26}$$

where σ is a constant often referred to as the kernel "width". The mapping function associated with the Gaussian kernel (often referred to as the radial basis function kernel) can be derived as follows (again, we follow the same approach as Shashua (2009)): without loss of generality, let $\sigma = 1$.

$$\kappa(\mathbf{x}, \mathbf{v}) = e^{\mathbf{x}^T\mathbf{v} - \frac{\|\mathbf{x}\|^2}{2} - \frac{\|\mathbf{v}\|^2}{2}} \tag{2.27}$$

$$\kappa(\mathbf{x}, \mathbf{v}) = e^{\mathbf{x}^T\mathbf{v}} e^{-\frac{\|\mathbf{x}\|^2}{2}} e^{-\frac{\|\mathbf{v}\|^2}{2}} \tag{2.28}$$

$$\kappa(\mathbf{x}, \mathbf{v}) = \sum_{j=0}^{\infty} \frac{\left(\mathbf{x}^T\mathbf{v}\right)^j}{j!} e^{-\frac{\|\mathbf{x}\|^2}{2}} e^{-\frac{\|\mathbf{v}\|^2}{2}} \tag{2.29}$$

$$\kappa(\mathbf{x}, \mathbf{v}) = \sum_{j=0}^{\infty} \left(\frac{e^{-\frac{\|\mathbf{x}\|^2}{2j}}}{\sqrt{j!}} \frac{e^{-\frac{\|\mathbf{v}\|^2}{2j}}}{\sqrt{j!}} \left(\mathbf{x}^T\mathbf{v}\right)^j\right) \tag{2.30}$$

$$\kappa(\mathbf{x}, \mathbf{v}) = \sum_{j=0}^{\infty} \sum_{|\mathbf{n}|=j} \left(\frac{e^{-\frac{\|\mathbf{x}\|^2}{2j}}}{\sqrt{j!}^{1/j}} \frac{e^{-\frac{\|\mathbf{v}\|^2}{2j}}}{\sqrt{j!}^{1/j}} \binom{j}{\mathbf{n}} \mathbf{x}^{\mathbf{n}}\mathbf{v}^{\mathbf{n}}\right) \tag{2.31}$$

where Equation (2.29) takes advantage of the Taylor series expansion of the exponential function: $e^x = \sum_{j=0}^{\infty} \frac{x^j}{j!}$ and Equation (2.31) uses the expansion of the polynomial kernel, given in Equation (2.21). By carefully collecting terms in Equation (2.31), the mapping function, $\Phi(\mathbf{x})$, can expressed as:

$$\Phi(\mathbf{x}) = \left(\left(\frac{e^{-\frac{\|\mathbf{x}\|^2}{2j}}}{\sqrt{j!}^{1/j}} \sqrt{\binom{j}{\mathbf{n}}} \mathbf{x}^{\mathbf{n}} \right)_{|\mathbf{n}|=j} \right)_{j=0,\cdots,\infty} \tag{2.32}$$

$$\Phi(\mathbf{x}) = \left(\frac{e^{-\frac{\|\mathbf{x}\|^2}{2j}}}{\sqrt{j!}^{1/j}} \sqrt{\binom{j}{\mathbf{n}}} \mathbf{x}^{\mathbf{n}} \right)_{j=0,\cdots,\infty,\ |\mathbf{n}|=j} . \tag{2.33}$$

Clearly, this expression is a little hard to parse. An alternative understanding of Equation (2.33) is as follows: for every value of $j \in 0, \cdots, \infty$, there are $\binom{m+j-1}{j}$ different possible values for $|\mathbf{n}| = j$. Therefore, the number of terms in $\Phi(\mathbf{x})$, M', is every possible value of $\mathbf{n}$ for every value of j. The terms in $\Phi(\mathbf{x})$ contains every combination of power of the terms within $\mathbf{x}$, i.e., $\prod_{\sum_{i=1}^{m} n_i = j} x_i^{n_i}$ multiplied by some scalar term. Given there are infinitely many values for j, there are clearly $M' = \infty$. To put another way, the Gaussian kernel represents a mapping of our input vector, $\mathbf{x}$ into an infinite dimensional feature space. In terms of scalability, calculating $\Phi(\mathbf{x})$ is of order $O(\infty)$ complexity, while calculating $\kappa(\mathbf{x}, \mathbf{v})$ is of order $O(m)$ complexity. The Gaussian kernel is often described as a "universal approximator" since it is able to approximate any continuous function with arbitrary accuracy (Hammer and Gersmann, 2003), this is a direct consequence of its mapping into infinite dimensional feature space.

Now that we've derived the mapping function of the Gaussian kernel, let's develop an intuitive understanding of how it can be used to model functions. To calculate the Gaussian kernel, the square of the Euclidean distance between $\mathbf{x}$ and $\mathbf{v}$ must be calculated. This value is then scaled, negated and finally the exponential is taken of the result. If $\mathbf{x} = \mathbf{v}$, then the distance between them is zero and the result of $\kappa(\mathbf{x}, \mathbf{v}) = 1$. If the distance between $\mathbf{x}$ and $\mathbf{v}$ is a large value, ω, and $\sigma = 1$, then the result of $\kappa(\mathbf{x}, \mathbf{v}) = e^{-\omega}$. As the distance between $\mathbf{x}$ and $\mathbf{v}$ approaches ∞, $\kappa(\mathbf{x}, \mathbf{v})$ approaches zero. Based on the above understanding, a common intuitive understanding of the Gaussian kernel as a measure of "similarity" between two vectors, $\mathbf{x}$ and $\mathbf{v}$. Figure 2.4 shows how linear regression in the dual space using the Gaussian kernel performs on the same data sets as Figure 2.2. Note that the dataset x_2 is modelled quite accurately by linear regression in the mapped space of the Gaussian kernel. The regression models shown in Figure 2.4 are remarkably similar to those found in Figure 2.3. The key difference is that for the models shown in Figure 2.3 we had to design our own mapping function. For complex problems, the design of such mapping functions is an open research problem and often significant domain knowledge is required to design them effectively. Alternatively, use of the Gaussian kernel requires only the selection of σ to be applied effectively to the problem at hand.

Finally, other popular kernel functions include the hyperbolic tangent kernel, $\kappa(\mathbf{x}_i, \mathbf{x}_j) = \tanh(\alpha \mathbf{x}_i^T \mathbf{x}_j + c)$, and the Laplacian kernel. The Laplacian kernel, $\kappa(\mathbf{x}_i, \mathbf{x}_j) = 2^{-\|\mathbf{x}_i - \mathbf{x}_j\|/\sigma}$, has gained some interest recently, since it can be implemented in hardware efficiently (Anguita et al., 2007). There have been kernel functions proposed for handling several different types of data, including strings (Lodhi et al., 2002) and graphs (Vishwanathan et al., 2010). For a good summary of several kernel functions and their applications, we refer readers to Souza (2010). Interestingly, despite not meeting all the requirements of a valid kernel function, the hyperbolic tangent kernel is reasonably popular. This is likely due to the hyperbolic tangent function:

a) being a popular activation function for neural networks; and

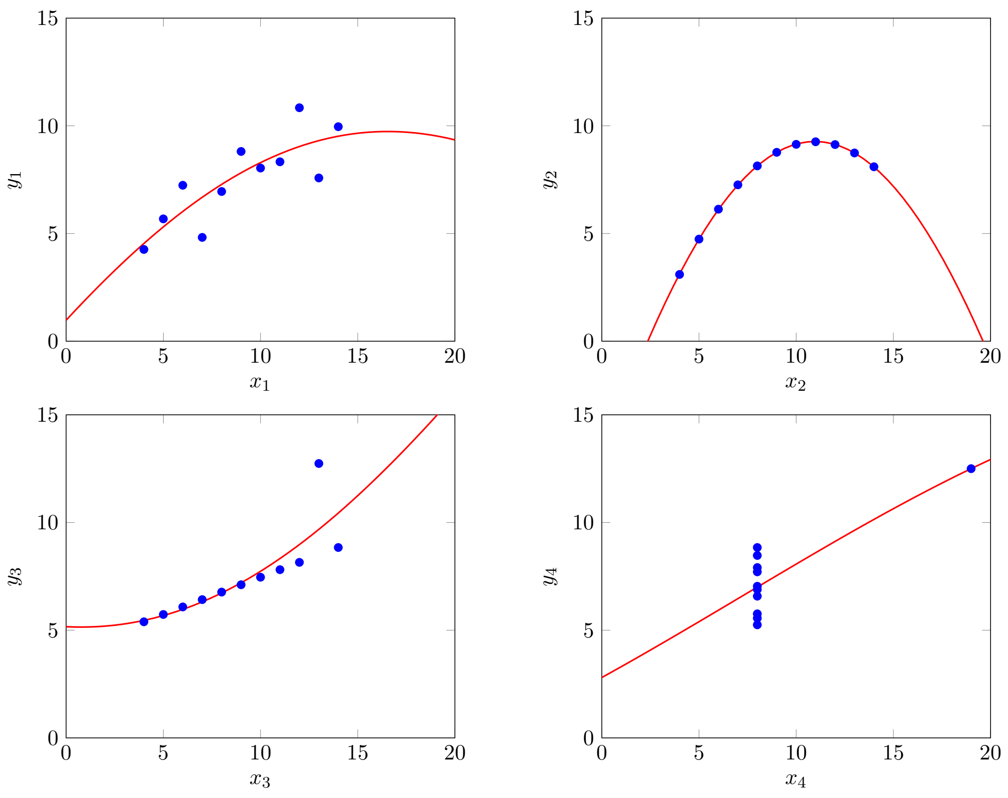


FIGURE 2.4: Anscombe's quartet - non-linear models created using linear regression in the space mapped by the Gaussian kernel.

b) working quite well in practice (Boughorbel, Tarel, and Boujemaa, 2005).

2.1.4 Properties of Kernel Functions

In Section 2.1.3, we discussed the dual representation and some mapping functions associated with popular kernel functions. We have shown that two popular kernel functions are equivalent to dot-products between two vectors in a space inferred by a mapping function, Φ. In this section, we discuss these kernel functions in a more theoretical way, i.e., the theory that underpins the examples provided in the previous section. We also detail several properties of kernel functions, and how to create new kernel functions through composition.

Firstly, we can see that from Equation (2.8) that the kernel matrix, $\mathbf{K}$, must be invertible, otherwise a solution for $\boldsymbol{\alpha}$ cannot be found. Explicitly, a chosen kernel function, κ, must be a positive-definite function, i.e., $\mathbf{K}$ is a positive semi-definite matrix. Explicitly, $\mathbf{K}$ is positive semi-definite if:

$$\mathbf{z}^T\mathbf{K}\mathbf{z} \geq 0 \ , \tag{2.34}$$

where $\mathbf{z} \in \mathbb{R}^N$. The proof is as follows (Domke, 2012a):

$$\mathbf{z}^T\mathbf{K}\mathbf{z} = \sum_{i=1}^{N}\sum_{j=1}^{N} z_i \kappa(\mathbf{x}_i, \mathbf{x}_j) z_j \tag{2.35}$$

$$\mathbf{z}^T\mathbf{K}\mathbf{z} = \sum_{i=1}^{N}\sum_{j=1}^{N} z_i \Phi(\mathbf{x}_i)^T \Phi(\mathbf{x}_j) z_j \tag{2.36}$$

$$\mathbf{z}^T\mathbf{K}\mathbf{z} = \sum_{i=1}^{N} z_i \Phi(\mathbf{x}_i)^T (\sum_{j=1}^{N} \Phi(\mathbf{x}_j) z_j) \tag{2.37}$$

$$\mathbf{z}^T\mathbf{K}\mathbf{z} = \left(\sum_{i=1}^{N} z_i \Phi(\mathbf{x}_i)\right)^T \left(\sum_{i=1}^{N} z_i \Phi(\mathbf{x}_i)\right) \tag{2.38}$$

$$\mathbf{z}^T\mathbf{K}\mathbf{z} = \left\|\sum_{i=1}^{N} z_i \Phi(\mathbf{x}_i)\right\|^2 \tag{2.39}$$

$$\mathbf{z}^T\mathbf{K}\mathbf{z} \geq 0 \ , \tag{2.40}$$

where $\mathbf{z} = [z_1, \cdots, z_N]$. Equation (2.40) shows us that if $\kappa(\mathbf{x}_i, \mathbf{x}_j) = \Phi(\mathbf{x}_i)^T\Phi(\mathbf{x}_j)$ then $\mathbf{K}$ is positive semi-definite. Mercer's theorem (Aronszajn, 1950) states that the opposite is also true, i.e., if κ is a continuous positive semi-definite kernel on a compact set $\mathcal{X}$, then there is an orthonormal basis of eigenfunctions, $\{\varphi_i\}$, such that the corresponding eigenvalues, $\{\lambda_i\}$, are non-negative. The eigenfunctions which correspond to non-zero eigenvectors are continuous on $\mathcal{X}$ and $\kappa(\mathbf{x}_i, \mathbf{x}_j)$ can be represented as follows:

$$\kappa(\mathbf{x}_i, \mathbf{x}_j) = \sum_{i=1}^{\infty} \lambda_i \varphi_i(\mathbf{x}_i) \varphi_i(\mathbf{x}_j) \ , \tag{2.41}$$

where $\mathbf{x}_i, \mathbf{x}_j \in \mathcal{X}$. Note, that $\mathcal{X}$ is a compact set, and therefore refers to any possible training example, $\mathbf{x}_i$. It is possible, that for a limited dataset that a $\mathbf{K}$ is positive semi-definite for a non-valid kernel function, but it is not guaranteed for all possible examples in the dataset. A full proof of this theorem is beyond the scope of this thesis, we recommend Bartlett (2008) and Daumé III (2004) for those interested readers.

2.1.5 Overfitting and Regularisation

As shown in Section 2.1.3, many kernel functions map the input data to a much higher (possibly infinite) feature space. Given this fact, it is common that kernel regression algorithms create complex models which effectively model the training data, but do not generalise well when applied to unseen data. This undesirable trait is referred to as *overfitting* and it's often necessary to employ several measures to avoid it. In this section, we'll discuss two methods for tackling this problem: *cross-validation* and *regularisation*. However, before we delve into these subjects, let us first look at an example of overfitting.

Let us again consider the dataset in Figure 2.4 and the Gaussian kernel function, given in Equation (2.26). To create the models in Figure 2.4, we used $\sigma = 32$ while performing kernel regression using approximate linear dependency kernel recursive least squares (ALD-KRLS) (Engel, Mannor, and Meir, 2004). We'll discuss KRLS in further detail in Section 2.1.6, for now consider it an algorithm for solving Equation (2.7). Our choice of $\sigma = 32$ was arbitrary, and on inspection it seemed to work

Dataset	MSE (f_1)	MSE (f_{16})	MSE (f_{32})	MSE (f_{64})
Anscombe(1)	4.45×10^{-30}	1.19	1.17	1.25
Anscombe(2)	8.79×10^{-31}	4.04×10^{-4}	1.10×10^{-4}	1.23
Anscombe(3)	3.59×10^{-30}	1.18	1.18	1.25
Anscombe(4)	1.25	1.25	1.25	1.25

TABLE 2.1: Modelling accuracy of different configurations of Gaussian kernels on Anscombe's quartet

well, so no further thought was given to this value. However, in real scenarios our training data might be very high-dimensional and not easy to plot and inspect. In this more common situation, we often have to rely on measurements to understand the quality of our model.

Let's again look at the data and models in Figure 2.4 and compare the mean squared error (MSE) between the dataset and the predicted values. Specifically, for each dataset we'll calculate: $MSE = \frac{1}{N}\sum_{i=1}^{N}(y_i - f_\sigma(x_i))^2$, where i is the i^{th} value in the given dataset and f_σ is the function learned by training ALD-KRLS using σ as the parameter in the Gaussian kernel. Table 2.1 shows the effect that changing σ has on the accuracy of modelling each of the Anscombe's quartet datasets. Clearly, in this table f_1 outperforms all other models. It achieves significantly lower modelling error on the first 3 datasets, while performing equally as well as the others on the 4^{th} dataset. Let's now plot all of these models together to see how they behave. Figure 2.5 shows the models found by f_1, f_{16}, f_{32} and f_{64} when trained on each of the

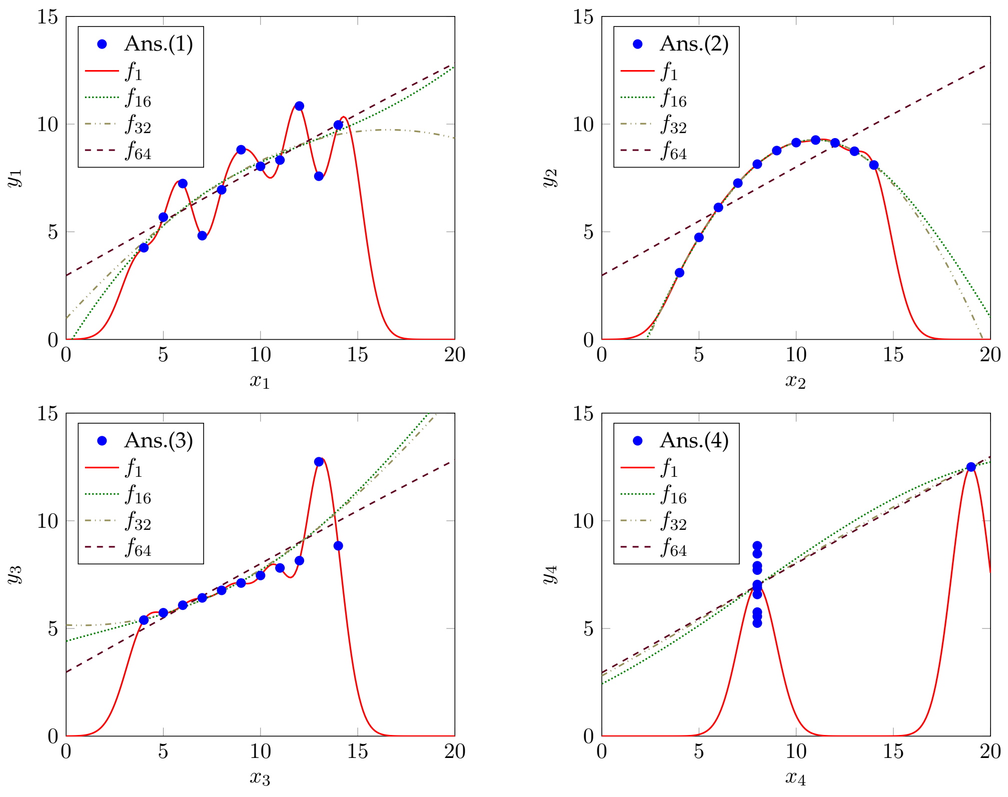


FIGURE 2.5: Anscombe's quartet - non-linear models created using linear regression in the space mapped by the Gaussian kernel using different values of σ.

Dataset	MSE (f_1)	MSE (f_{16})	MSE (f_{32})	MSE (f_{64})
Anscombe(1)	10.96	2.36	1.99	1.86
Anscombe(2)	1.48	0.0013	0.0075	2.17
Anscombe(3)	2.97	4.16	3.09	2.15
Anscombe(4)	15.75	5.96	4.71	4.40

TABLE 2.2: Validation error of different configurations of Gaussian kernels on Anscombe's quartet

datasets shown in each figure. Interestingly, we can now see why f_1 achieves such low modelling error, it's able to interpolate between the points of datasets Ans. 1-3 very well. By inspection of Equation (2.26), we can quickly understand why smaller values of σ can model more complex datasets, as $\sigma \to 0$, $\kappa(\mathbf{x}_i, \mathbf{x}_j) \to 0$ when $\mathbf{x}_i \neq \mathbf{x}_j$. When $\mathbf{x}_i = \mathbf{x}_j$, $\kappa(\mathbf{x}_i, \mathbf{x}_j) = 1$. Intuitively, we can think σ as a tuning parameter, allowing $\kappa(\mathbf{x}_i, \mathbf{x}_j)$ to model more complex and varying functions when σ is small, but possibly at the cost of generality. For example, let's consider Ans. (1) in Figure 2.5, f_1 has found a significantly more complex model than f_{32} and because of that, it is able to model each of the points in the dataset very well. However, what if the training data is corrupted by some noise? Or what if x_1 does not capture all of the information necessary to accurately predict y_1? In either of these cases, it's possible that f_1 performs worse than f_{32} on unseen data, despite being more accurate on the training set. In this instance, we refer to f_1 as overfitting to the training data.

Cross-validation

Cross-validation is an important, but relatively basic technique to try to prevent overfitting. The key idea is that while a data scientist is selecting the hyperparameters (e.g., choice of κ, σ, etc.), they split the training data into two subsets: the training set and the validation set. The validation set is not used at all to train the model, it's only used to evaluate the accuracy of the model. The training hyperparameters which perform best on the validation set will then be selected to train the model.

The purpose of the validation set is to simulate how a training algorithm or a trained model will perform on unseen data, which represents a better estimate of how these will be used under deployment.

There are several types of cross-validation methods, for an introduction to the subject we recommend Domke (2012b). In this section, we briefly describe k-fold cross-validation, as the technique is leveraged several times in subsequent chapters in order to tune training hyperparameters. In k-fold cross validation, we partition the original training data up into k randomly sampled, equally sized datasets. We then train and validate our training hyperparameters k times, each time using a different subset of our original dataset as the validation set, while using the other $(k-1)/k$ subsets as the training set. In most cases, the overall accuracy of the technique is then determined by using the average validation score across the k runs.

Let's now use this approach to update Table 2.1 to show validation error, rather than training error. Table 2.2, the average validation error for k-fold cross-validation of each of the datasets when $k = 11$. Note, that since the entire training dataset contains 11 values, i.e., $N = 11$, this type of cross-validation is also known as leave-one-out cross-validation (LOOCV). Comparing Tables 2.1 and 2.2, we get a very different story after cross-validation. In Table 2.1, f_1 performed equally well, or outperformed every other model, While in Table 2.2, f_1 does not perform the best in any dataset in Anscombe's quartet. For Ans. (1-3), f_1 achieves a $MSE > 10^{29}$ higher during cross-validation, than when simply testing on the training set. Conversely, f_{64}, which was preferred for no dataset in Table 2.1, now outperforms all others for datasets Ans. (1,3,4), while f_{16} outperforms the others for Ans. (2). Note, that for all models (f_σ) and datasets the cross-validation MSE shown in Table 2.2 is higher than the MSE shown in Table 2.2. This shows that all models trained in Table 2.1 did overfit somewhat to the training data.

In reality, we have no way of *knowing* which model will generalise the best to unseen data, in particular Ans. (1) in Figure 2.5. However, we suggest that in the absence of evidence to support higher model complexity, we should prefer models of lower complexity. Again, with reference to Ans. (1) and the models in Figure 2.5, more training data would be required to prefer f_1 over any of the other models. Prior works make similar reasoning about model selection, with some likening this preference for simpler models to an application of Occam's Razor to model selection (Nannen, 2003).

Regularisation

Regularisation is a term used to describe a group of methods which are used to solve ill-posed problems or to prevent overfitting. Regularisation methods achieve this by adding extra constraints or objectives to the cost function of the problem, to prefer certain solutions over others. Similar to cross-validation, it is often used to promote simpler models over more complex models. Regularisation can be used to promote dense solutions (Tikhonov, 1943), sparse solutions (Gorodnitsky and Rao, 1997) and various combinations and complex patterns on our solutions (Yuan and Lin, 2006). In this work we look at L2-regularisation, a specific case of Tikhonov regularisation (Tikhonov, 1943) (otherwise known as weight decay), a common form of regularisation which is used in this work.

When L2-regularisation is introduced, the cost function (provided in Equation (2.4)) becomes:

$$J'' = \min_{\tilde{\mathbf{h}}} \left(\left\| \mathbf{y} - \tilde{\mathbf{X}}\tilde{\mathbf{h}} \right\|^2 + \lambda \left\| \tilde{\mathbf{h}} \right\|^2 \right) \tag{2.42}$$

$$J'' = \min_{\boldsymbol{\alpha}} \left(\| \mathbf{y} - \mathbf{K}\boldsymbol{\alpha} \|^2 + \lambda \boldsymbol{\alpha}^T \mathbf{K} \boldsymbol{\alpha} \right) \quad , \tag{2.43}$$

where $\lambda \in \mathbb{R}^+$ is the regularisation term and in Equation (2.43) we invoke the property that $\tilde{\mathbf{h}} = \tilde{\mathbf{X}}^T \boldsymbol{\alpha}$. In the primal formulation in Equation (2.42), this is referred to as ridge regression, while in the dual, kernelised formulation in Equation (2.43), this is referred to as kernel ridge regression (Shawe-Taylor and Cristianini, 2004). With regularisation, the ideal solution of weights becomes:

$$\boldsymbol{\alpha} = (\mathbf{K} + \lambda \mathbf{I})^{-1} \mathbf{y} \tag{2.44}$$

where $\mathbf{I}$ is the identity matrix.

Training a model with L2-regularisation has multiple benefits:

a) ill-posed problems can now be solved, for example if $\mathbf{K}$ is singular;

b) trained models tend to generalise better, perhaps due to the equivalence between L2-regularisation and noise injection (Bishop, 1995);

c) it reduces an algorithm's sensitivity to finite precision effects, which we discuss in more detail in Section 2.3.

The choice of λ is often dataset and algorithm specific, as such it is usually one of the hyperparameters chosen during cross-validation.

2.1.6 Online Machine Learning

In previous sections, we've discussed linear regression, non-linear regression and kernel regression. In this section, we discuss a class of problems know as *online problems* and classes of algorithms, known as adaptive filters (Haykin, 2005), which attempt to create regressive models of online problems. For an online problem, not all training data is known in advance and therefore a new solution must be recalculated as new observations are provided. As such, techniques like batch gradient will often perform very badly as the entire training set will often need to be iterated through several times in order find a solution for $\mathbf{h}$. Another approach known as SGD provides a more efficient means to estimate $\mathbf{h}$. For least-squares regression problems, like ones trying to find the solution for the cost function in Equation (2.1), this approximation to BGD becomes equivalent to the least mean squares (LMS) algorithm (Widrow and Hoff, 1960).

The Least Mean Squares Algorithm The LMS algorithm is a very simple and useful linear adaptive filter. It updates the solution vector, h, for each new sample using the following method:

$$\mathbf{h}_n = \mathbf{h}_{n-1} + \eta \left(y_n - \mathbf{x}_n^T \mathbf{h}_{n-1} \right) \mathbf{x}_n \quad , \tag{2.45}$$

where $\mathbf{x}_n$ is the latest input vector, y_n is the desired output, $\mathbf{h}_n$ is the n^{th} estimate of the optimal solution for weights and η is the step size parameter. Also, $\mathbf{h}_0 = \mathbf{0}$. Equation (2.45) is comprised of the following: the a priori prediction,

$$\tilde{y}_n = \mathbf{x}^T \mathbf{h}_{n-1} \quad , \tag{2.46}$$

and the estimation error,

$$e_n = y_n - \tilde{y}_n \quad , \tag{2.47}$$

The LMS algorithm makes a small, incremental update to the current solution vector, $\mathbf{h}_{n-1}$, which reduces the *instantaneous error* in the a priori prediction, i.e., e_n. LMS is easy to compute but it does not provide an optimal solution for $\mathbf{h}_n$, rather it updates its estimate based on the amount prediction error for the n^{th} training sample. In practice, the solution often oscillates in the space around the optimal weights. This property means that the prediction capability of algorithms based around SGD, such as the LMS algorithm, is often impaired slightly, though it does also have the effect of allowing algorithms to track non-stationary systems.

Recursive Least Squares (RLS) For online problems where an optimal solution for $\mathbf{h}$ is preferred, the recursive least squares (RLS) algorithm provides a direct and computationally efficient solution for $\mathbf{h}$ for every new sample. RLS achieves this without needing to fully recalculate the matrix inverse, from Equation (2.3). It uses the previously known value of $(\mathbf{X}^T\mathbf{X})^{-1}$ to calculate the next value when a new input/output pair is received using the *matrix inversion lemma* (Woodbury, 1950), given in Equation (2.48). This provides a recursive solution for $(\mathbf{X}^T\mathbf{X})^{-1}$ with a computational complexity of $O(M^2)$.

$$\left(\mathbf{A}+\mathbf{x}_n\mathbf{x}_n^T\right)^{-1}=\mathbf{A}^{-1}-\frac{\mathbf{A}^{-1}\mathbf{x}_n\mathbf{x}_n^T\mathbf{A}^{-1}}{1+\mathbf{x}_n^T\mathbf{A}^{-1}\mathbf{x}_n} \tag{2.48}$$

where $\mathbf{x}_n$ is the latest vector of observations and $\mathbf{A}=\mathbf{X}_{n-1}^T\mathbf{X}_{n-1}$ where $\mathbf{X}_{n-1}$ is given by $\mathbf{X}_{n-1}=\{\mathbf{x}_1,\mathbf{x}_2,\cdots,\mathbf{x}_{n-1}\}$.

The RLS algorithm is designed to calculate recursive updates for the inverse covariance matrix, $\mathbf{P}=\mathbf{R}_{\mathbf{x}}^{-1}=\left(\mathbf{X}^T\mathbf{X}\right)^{-1}$, and the change in the cross-covariance, $\mathbf{r}_{\mathbf{x}y}=\mathbf{X}^Ty$, while trying to reuse as much computation is possible. In order to calculate an update to the model found by the RLS algorithm, given a new input/output pair, $\mathbf{x}_n/y_n$, we first calculate the *gain vector*, $\mathbf{g}_n$, as follows:

$$\mathbf{g}_n=\frac{\mathbf{P}_{n-1}\mathbf{x}_n}{\zeta+\mathbf{x}_n^T\mathbf{P}_{n-1}\mathbf{x}_n}\quad, \tag{2.49}$$

where $0<\zeta\geq 1$ is the *forgetting factor*, a parameter providing recent samples an exponentially higher weighting than old samples, and $\mathbf{P}_{n-1}$ is the inverse covariance matrix after sample $n-1$.

$$\mathbf{P}_n=\zeta^{-1}\mathbf{P}_{n-1}-\mathbf{g}_n\mathbf{x}_n^T\zeta^{-1}\mathbf{P}_{n-1}\quad. \tag{2.50}$$

Finally, the parameter update, $\mathbf{h}_n$, is calculated as follows:

$$\mathbf{h}_n=\mathbf{h}_{n-1}+e_n\mathbf{g}_n\quad, \tag{2.51}$$

Where e_n is estimation error given in Equation (2.47). Algorithm 2.2 shows pseu-

Algorithm 2.2 Pseudocode: a training step for the RLS algorithm.

```
Select values for the regularisation parameter, $\lambda$, and the forgetting factor, $\zeta$.
Initialise $\mathbf{P}_0$ as $\lambda^{-1}\mathbf{I}$ and $\mathbf{h}_0$ as $\mathbf{0}$.
for $n = 1, 2, \cdots$ do
    Calculate the a priori prediction, $\tilde{y}_n$, using Equation (2.46)
    Calculate prediction error, $e_n$, using Equation (2.47)
    Calculate the gain vector, $\mathbf{g}_n$, using Equation (2.49)
    Update the inverse covariance matrix, $\mathbf{P}_n$, using Equation (2.50)
    Calculate the updated solution vector, $\mathbf{h}_n$, using Equation (2.51)
end for
```

docode for the RLS with regularisation and exponential weighting. The exponential weighting means that over time, older examples are progressively forgotten. Despite the common term for ζ being a "forgetting factor", it's easier to think of it as a *remembering factor*. If $\zeta=1$, prior examples are effectively remembered without any weighting applied. Otherwise, ζ determines a multiplicative amount by which each older example is remembered when a new example arrives. Specifically, the weighting that's applied to $(\mathbf{x}_1,y_1)$ when $(\mathbf{x}_2,y_2)$ arrives is ζ. When $(\mathbf{x}_3,y_3)$ arrives, $(\mathbf{x}_1,y_1)$ will be weighted by ζ^2 and $(\mathbf{x}_2,y_2)$ by ζ. Generalising this, when the n^{th} example arrives, the weighting, ω, that's applied to the $\tilde{n}^{th}$ example, where $0\geq\tilde{n}\geq n$, is given by: $\omega=\zeta^{n-\tilde{n}}$. The effect of this is that, similar to the LMS algorithm, the RLS algorithm with exponential weighting can track non-stationary systems.

Kernel Adaptive Filtering

Now that we've looked at some standard adaptive filters, specifically the LMS algorithm and the RLS algorithm, we can study their kernel-based equivalents: kernel adaptive filters (KAFs). Specifically in this section, we look at the equivalent of the LMS, the kernel least mean squares (KLMS) algorithm (Pokharel, Liu, and Principe, 2007; Liu, Pokharel, and Principe, 2008). Furthermore, we discuss variants of kernel-based versions of RLS, in particular, the KRLS algorithm (Engel, Mannor, and Meir, 2004; Liu, Pokharel, and Principe, 2008) and the sliding window kernel recursive least squares (SW-KRLS) algorithm (Van Vaerenbergh, Via, and Santamaria, 2006).

The Kernel Least Mean Squares Algorithm The KLMS algorithm is a formulation of LMS in the dual representation, which then allows us to utilise a kernel function, κ. Recalling Equations (2.45) to (2.47) which define the LMS algorithm, specifically, the prediction function, the estimation error and the weight update function. Firstly, let us write equivalent variants of the above function, but with the introduction of the mapping function, Φ. The weight update function for the n^{th} input/output pair, $(\mathbf{x}_n, y_n)$, becomes:

$$\tilde{\mathbf{h}}_n = \tilde{\mathbf{h}}_{n-1} + \eta \left(y_n - \Phi(\mathbf{x}_n)^T \tilde{\mathbf{h}}_{n-1} \right) \Phi(\mathbf{x}_n) \ , \tag{2.52}$$

where again, η is the step-size parameter, and $\tilde{\mathbf{h}}_n$ is the n^{th} estimate of the solution vector. The problem with this formulation, as we motivated in Section 2.1.3, is that if we want Φ to map our input vectors to a very high dimensional feature, e.g. $M' \to \infty$, it might not be feasible to calculate $\Phi(\mathbf{x}_n)$ or to store or calculate $\tilde{\mathbf{h}}_n$. Given this, we need to reformulate the LMS algorithm. Firstly, let's consider the a priori prediction function at time 2, given by:

$$\tilde{y}_2 = \Phi(\mathbf{x}_2)^T \tilde{\mathbf{h}}_1 \tag{2.53}$$
$$\tilde{y}_2 = \Phi(\mathbf{x}_2)^T \left(\tilde{h}_0 + \eta \left(\eta e_1 \Phi(\mathbf{x}_1) \right) \right) \tag{2.54}$$
$$\tilde{y}_2 = \eta e_1 \Phi(\mathbf{x}_2)^T \Phi(\mathbf{x}_1) \tag{2.55}$$
$$\tilde{y}_2 = \eta e_1 \kappa(\mathbf{x}_2, \mathbf{x}_1) \tag{2.56}$$
$$\tilde{y}_2 = \alpha_1 \kappa(\mathbf{x}_2, \mathbf{x}_1) \ , \tag{2.57}$$

where e_1 is the a priori prediction error for example $(\mathbf{x}_1, y_1)$, which is:

$$e_1 = y_1 - \Phi(\mathbf{x}_1)^T \tilde{\mathbf{h}}_0 \tag{2.58}$$
$$e_1 = y_1 \ , \tag{2.59}$$

in Equation (2.54) we substitute in Equation (2.52) for $\tilde{\mathbf{h}}_1$, in Equations (2.55) and (2.59) we use the fact that $\tilde{\mathbf{h}}_0 = \mathbf{0}$ and $\alpha_1 = \eta e_1$.

Now the we have an expression for $\tilde{y}_2$, we can use this to calculate $e_2 = y_2 - \tilde{y}_2$ and $\alpha_2 = \eta e_2$. We can subsequently use these expressions to find an expression for the a priori prediction for example $(\mathbf{x}_n, y_n)$, given by:

$$\tilde{y}_n = \sum_{i=1}^{n-1} \alpha_i \kappa(\mathbf{x}_n, \mathbf{x}_i) \ , \tag{2.60}$$

where

$$\alpha_n = \eta e_n \ , \tag{2.61}$$

where

$$e_n = y_n - \tilde{y}_n \quad . \tag{2.62}$$

At first these equations appear to have a circular dependency, but note that $\tilde{y}_n$ only depends on the calculations of $\{\tilde{y}_1, \cdots, \tilde{y}_{n-1}\}$, and that $\tilde{y}_1 = 0$. Pseudocode for the

Algorithm 2.3 Pseudocode: a training step for the KLMS algorithm.

```
Select a value for the step-size parameter, $\eta$, and a kernel function, $\kappa$.
Initialise $\boldsymbol{\alpha}_0$ as an empty vector, i.e., $\boldsymbol{\alpha}_0 = [\,]$.
for $n = 1, 2, \cdots$ do
    Calculate the a priori prediction, $\tilde{y}_n$, using Equation (2.60)
    Calculate prediction error, $e_n$, using Equation (2.62)
    Calculate the updated weight vector using $\boldsymbol{\alpha}_n = [\boldsymbol{\alpha}_{n-1}^T, \alpha_n]^T$ and Equation (2.61)
end for
```

KLMS algorithm is provided in Algorithm 2.3. Note, in the pseudocode example, we define a weight vector $\boldsymbol{\alpha}_n$, which is slowly appended with each value of α_n. At this point, we have derived a kernel-based version of LMS, known as KLMS. Although LMS is a very popular and useful adaptive filter, we can see that KLMS has an issue which limits its use in online settings. Specifically, the calculation of Equation (2.60) depends on a kernel evaluation between the current example, $\mathbf{x}_n$, and all previous examples, $\{\mathbf{x}_1, \cdots, \mathbf{x}_{n-1}\}$, resulting in a linear increase in memory usage and computational time for each new input/output pair, $O(nm)$ in both memory and computational requirements. This means that KLMS may not be suitable for applications where:

a) there are strict throughput requirements (e.g., there is a specific sample rate which must be maintained);

b) there are tight memory restrictions (e.g., in an embedded system); and

c) the algorithm is expected to run for long periods of time or indefinitely.

We will address these issues in a subsequent section, for now we wish to show that we can define a kernel-based version of LMS which is mathematically exactly equivalent to the original, but with the inclusion of a mapping to some high dimensional feature space via a kernel function.

Kernel Recursive Least Squares A similar technique can be applied to the RLS algorithm resulting in the KRLS algorithm (Engel, Mannor, and Meir, 2004; Liu, Príncipe, and Haykin, 2011). We start with a simple version of KRLS, as described by Liu, Príncipe, and Haykin (2011), then we move onto more practical, but algorithmically complex versions, described by Engel, Mannor, and Meir (2004), Van Vaerenbergh, Via, and Santamaria (2006) and Van Vaerenbergh et al. (2010). Firstly, we start with the dual formulation and solution to the regularised least squares problem, given in Equations (2.43) and (2.44) respectively. We briefly reiterate these equations below. The dual formulation of the least squares problem is given by:

$$J'' = \min_{\boldsymbol{\alpha}} \left(\|\mathbf{y} - \mathbf{K}\boldsymbol{\alpha}\|^2 + \lambda \boldsymbol{\alpha}^T \mathbf{K} \boldsymbol{\alpha} \right) \quad , \tag{2.63}$$

where $K_{i,j}$, the entry in the i^{th} row and j^{th} column of $\mathbf{K}$, is given by $\kappa(\mathbf{x}_i, \mathbf{x}_j)$, $\mathbf{y} = [y_1, \cdots, y_n]^T$, λ is the regularisation term, and $\boldsymbol{\alpha} \in \mathbb{R}^n$ is the solution vector. The ideal solution is given by:

$$\boldsymbol{\alpha} = (\mathbf{K} + \lambda \mathbf{I})^{-1} \mathbf{y} \tag{2.64}$$

Similar to RLS, the goal of KRLS is to be able to efficient find an updated version of $\boldsymbol{\alpha}$. Given a solution for $\boldsymbol{\alpha}_{n-1}$ and $\mathbf{K}_{n-1}^{-1}$, we need to find $\boldsymbol{\alpha}_n$ and $\mathbf{K}_n^{-1}$ given $(\mathbf{x}_n, y_n)$, where $\boldsymbol{\alpha}_n / \mathbf{K}_n^{-1}$ is the optimal weight vector / inverse kernel matrix respectively after receiving the n^{th} example. Firstly, let's note that $\mathbf{K}_n$ becomes:

$$\mathbf{K}_n = \begin{bmatrix} \mathbf{K}_{n-1} & \mathbf{k}_n \\ \mathbf{k}_n^T & \kappa(\mathbf{x}_n, \mathbf{x}_n) + \lambda \end{bmatrix} \ , \tag{2.65}$$

where $\mathbf{k}_n$ is the kernel vector, given by:

$$\mathbf{k}_n = [\kappa(\mathbf{x}_n, \mathbf{x}_1), \cdots, \kappa(\mathbf{x}_n, \mathbf{x}_{n-1})] \ . \tag{2.66}$$

Given this observation, it's clear we can use the block matrix inversion identity (Wolf, 1978) to find an efficient solution for $\mathbf{K}_n^{-1}$. There are several ways of expressing the block matrix identity, we choose the formulation which relies on the fewest calculations as possible, as follows:

$$\begin{bmatrix} \mathbf{A} & \mathbf{B} \\ \mathbf{C} & \mathbf{D} \end{bmatrix}^{-1} = \begin{bmatrix} \mathbf{A}^{-1} + \mathbf{A}^{-1}\mathbf{B}\left(\mathbf{D} - \mathbf{C}\mathbf{A}^{-1}\mathbf{B}\right)^{-1}\mathbf{C}\mathbf{A}^{-1} & -\mathbf{A}^{-1}\mathbf{B}\left(\mathbf{D} - \mathbf{C}\mathbf{A}^{-1}\mathbf{B}\right)^{-1} \\ -\left(\mathbf{D} - \mathbf{C}\mathbf{A}^{-1}\mathbf{B}\right)^{-1}\mathbf{C}\mathbf{A}^{-1} & -\left(\mathbf{D} - \mathbf{C}\mathbf{A}^{-1}\mathbf{B}\right)^{-1} \end{bmatrix} \ , \tag{2.67}$$

where $\mathbf{A}$, $\mathbf{B}$, $\mathbf{C}$, $\mathbf{D}$ are arbitrarily sized sub-matrices and both $\mathbf{A}$ and $(\mathbf{D} - \mathbf{C}\mathbf{A}^{-1}\mathbf{B})$ are invertible. Relating Equation (2.65) and Equation (2.67), we can see that: $\mathbf{A} = \mathbf{K}$, $\mathbf{B} = \mathbf{k}_n$, $\mathbf{C} = \mathbf{k}_n^T$ and $\mathbf{D} = \kappa(\mathbf{x}_n, \mathbf{x}_n) + \lambda$. Defining $\mathbf{z}_n$ and r_n as follows:

$$\mathbf{z}_n = \mathbf{K}_{n-1}^{-1}\mathbf{k}_n \ , \tag{2.68}$$

and

$$r_n = \lambda + \kappa(\mathbf{x}_n, \mathbf{x}_n) - \mathbf{z}_n^T \mathbf{k}_n \ , \tag{2.69}$$

We get the update expression for the inverse kernel matrix, $\mathbf{K}_n^{-1}$, as follows:

$$\mathbf{K}_n^{-1} = \frac{1}{r_n} \begin{bmatrix} r_n \mathbf{K}_{n-1}^{-1} + \mathbf{z}_n \mathbf{z}_n^T & -\mathbf{z}_n \\ -\mathbf{z}_n^T & 1 \end{bmatrix} \ . \tag{2.70}$$

Now that the inverse kernel matrix, $\mathbf{K}_n^{-1}$ has been calculated, we can calculate the updated weight vector, $\boldsymbol{\alpha}_n$, as follows:

$$\boldsymbol{\alpha}_n = \mathbf{K}_n^{-1}\mathbf{y}_n \tag{2.71}$$

$$\boldsymbol{\alpha}_n = \begin{bmatrix} \mathbf{K}_{n-1}^{-1} + r_n^{-1}\mathbf{z}_n\mathbf{z}_n^T & -r_n^{-1}\mathbf{z}_n \\ -r_n^{-1}\mathbf{z}_n^T & r_n^{-1} \end{bmatrix} \begin{bmatrix} \mathbf{y}_{n-1} \\ y_n \end{bmatrix} \tag{2.72}$$

$$\boldsymbol{\alpha}_n = \begin{bmatrix} \boldsymbol{\alpha}_{n-1} \\ 0 \end{bmatrix} + r_n^{-1} \begin{bmatrix} \mathbf{z}_n\mathbf{z}_n^T & -\mathbf{z}_n \\ -\mathbf{z}_n^T & 1 \end{bmatrix} \begin{bmatrix} \mathbf{y}_{n-1} \\ y_n \end{bmatrix} \tag{2.73}$$

$$\boldsymbol{\alpha}_n = \begin{bmatrix} \boldsymbol{\alpha}_{n-1} \\ 0 \end{bmatrix} + r_n^{-1} \begin{bmatrix} \mathbf{z}_n \tilde{y}_n \\ -\tilde{y}_n \end{bmatrix} + r_n^{-1} \begin{bmatrix} -\mathbf{z}_n y_n \\ y_n \end{bmatrix} \tag{2.74}$$

$$\boldsymbol{\alpha}_n = \begin{bmatrix} \boldsymbol{\alpha}_{n-1} - r_n^{-1} e_n \mathbf{z}_n \\ r_n^{-1} e_n \end{bmatrix} \ , \tag{2.75}$$

where $\mathbf{y}_n = [y_1, \cdots, y_n]$, $\tilde{y}_n$ is the a priori prediction, given by:

$$\tilde{y}_n = \mathbf{k}_n^T \boldsymbol{\alpha}_{n-1} \quad , \tag{2.76}$$

e_n is the prediction error: $e_n = y_n - \tilde{y}_n$, and in Equation (2.74) we take advantage of the following:

$$\mathbf{z}^T \mathbf{y}_{n-1} = \left(\mathbf{K}_{n-1}^{-1}\mathbf{k}_n\right)^T \mathbf{y}_{n-1} \tag{2.77}$$

$$\mathbf{z}^T \mathbf{y}_{n-1} = \mathbf{k}_n^T \mathbf{K}_{n-1}^{-1} \mathbf{y}_{n-1} \tag{2.78}$$

$$\mathbf{z}^T \mathbf{y}_{n-1} = \mathbf{k}_n^T \boldsymbol{\alpha}_{n-1} \tag{2.79}$$

$$\mathbf{z}^T \mathbf{y}_{n-1} = \tilde{y}_n \quad , \tag{2.80}$$

where in Equation (2.79) we substitute in Equation (2.64), and in Equation (2.80) we substitute in Equation (2.76). Finally, pseudocode for the KRLS algorithm is provided

Algorithm 2.4 Pseudocode: a training step for the KRLS algorithm.

```
Select a value for the regularisation parameter, $\lambda$, and a kernel function, $\kappa$.
Initialise $\mathbf{K}_1^{-1} = \frac{1}{\lambda + \kappa(\mathbf{x}_1, \mathbf{x}_1)}$ and $\boldsymbol{\alpha}_1 = \mathbf{K}_1^{-1} y_1$.
for $n = 2, 3, \cdots$ do
    Calculate the kernel vector, $\mathbf{k}_n$, using Equation (2.66)
    Calculate $\mathbf{z}_n$ using Equation (2.68)
    Calculate $r_n$ using Equation (2.69)
    Calculate the a priori prediction as $\tilde{y}_n = \mathbf{k}_n^T \boldsymbol{\alpha}_{n-1}$
    Calculate the prediction error, $e_n$, as $e_n = y_n - \tilde{y}_n$
    Update the inverse kernel matrix, $\mathbf{K}_n^{-1}$, using Equation (2.70)
    Update the weight vector, $\boldsymbol{\alpha}_n$, using Equation (2.75)
end for
```

in Algorithm 2.4. Note, that the regularisation factor is handled by the initialisation of $\mathbf{K}_1^{-1}$ and the calculation of r_n using Equation (2.69).

Clearly, KRLS suffers from similar problems as KLMS, which limit its use in online settings. In the dual representation, the update step of the KRLS algorithm scales with $O(n^2 + nm)$ in both memory and computational requirements. When KRLS is used for online applications n will increase unbounded as the algorithm continues to train and makes predictions. Similar to KLMS, KRLS has properties which limit its use in online applications. Specifically, online applications which have any of the following requirements:

a) strict throughput requirements (e.g., there is a specific sample rate which must be maintained);

b) tight memory restrictions (e.g., in an embedded system); or

c) long or indefinite run times.

Furthermore, KRLS scales worse than KLMS, $O(n^2 + nm)$ versus $O(nm)$, though at every iteration KRLS will calculate an optimal, least squares solution for $\boldsymbol{\alpha}$, rather than converge towards the optimal solution, like KLMS does. In reality, it's a trade-off which will need to be made between accuracy and performance for each application. Finally, the length of the feature vector, m, is fixed while n increases with every new training example. Given this, the n^2 term will quickly dominate the memory and compute requirements for KRLS. Again, our aim here is not to the say that this formulation of KRLS is practical, rather to show that we can describe a kernel-based equivalent of the RLS algorithm. It should also be noted that an exponentially weighted variant of KRLS can also be derived, we do not describe this as it's not used in this work. For those interested, we recommend reading Liu, Príncipe, and Haykin (2011). In this next section, we look at methods to make these algorithms applicable to online applications.

Kernel Adaptive Filtering with Compact Dictionaries There are several methods for reducing the computational cost of both KLMS-like and KRLS-like algorithms. One of the key methods to achieve this, is to reduce the memory and computational costs associated with the kernel regression model itself. Recalling Equation (2.60), the a-priori prediction function for both KLMS and KRLS at time n is:

$$\tilde{y}_n = \sum_{i=1}^{n-1} \alpha_i \kappa(\mathbf{x}_n, \mathbf{x}_i) \quad . \tag{2.81}$$

The calculation of $\tilde{y}_n$ is dependent on every previous training example, $\mathbf{x}_1 \to \mathbf{x}_{n-1}$. In other words, the trained model is *parametrised* by all previously seen training examples. The consequence of this is that the model increases linearly over time in its computational and memory requirements. An alternative is to parameterise the model on a subset of training examples, known as a *dictionary*. Methods on how to decide which training examples are stored in the dictionary is an active area of research. For context, the dictionary in KAFs are very similar to the support vectors in support vector machine (SVM) literature. In Engel, Mannor, and Meir (2004), a sparsification procedure is employed to prevent any samples from being admitted to the dictionary if they can be represented by linear combinations of the previous samples. This helps to reduce the computational complexity but does not bound the computational requirements as would be required by a online, embedded solution with precise timing requirements. Previous works have bounded this computation cost including the SW-KRLS algorithm (Van Vaerenbergh, Via, and Santamaria, 2006), the fixed budget kernel recursive least squares (FB-KRLS) algorithm (Van Vaerenbergh et al., 2010) and the kernel recursive least squares tracker (KRLS-T) algorithm (Van Vaerenbergh, Lázaro-Gredilla, and Santamaría, 2012). These modifications to the KRLS bound the computation cost by setting a limit, N', to the number of training samples which can be stored in the dictionary. The variants on the KRLS algorithms are very similar at their core but have different techniques for determining which training samples get added/removed from the dictionary. The SW-KRLS algorithm is described in detail below as it provides good prediction/tracking capability while also being the simplest to implement in hardware. The SW-KRLS algorithm removes any training samples that are not in a fixed time window, N'. Given a stream of input/output pairs, $\{(x_1, y_1), (x_2, y_2), \cdots\}$, at training sample n, the input matrix becomes $\mathbf{X}_n = [x_n, x_{n-1}, \cdots, x_{n-N'+1}]$ and the output vector becomes $\mathbf{Y}_n = [y_n, y_{n-1}, \cdots, y_{n-N'+1}]$.

In order to calculate α_n, the n^{th} estimate of the weights, $\mathbf{K}_n^{-1}$, the inverse kernel matrix can be calculated using $\mathbf{K}_{n-1}^{-1}$ and $\mathbf{K}_n$, the n^{th} kernel matrix. $\mathbf{K}_n$ can be calculated as follows:

$$\hat{\mathbf{K}}_n = \begin{bmatrix} \mathbf{K}_{n-1} & k_n(x_n) \\ k_n(x_n)^T & k_{nn} + c \end{bmatrix} \tag{2.82}$$

where $k_n(x_n) = [\kappa(x_{n-N'+1}, x_n), \cdots, \kappa(x_{n-1}, x_n)]^T$, $k_{nn} = \kappa(x_n, x_n)$, c is a regularisation constant, discussed in the following subsection, and $\mathbf{K}_{n-1}$ is the kernel matrix calculated from the previous training sample. $\mathbf{K}_n$ is defined as follows:

$$\mathbf{K}_n = \begin{bmatrix} k_{n-N,n-N} + c & p^T \\ p & \hat{\mathbf{K}}_{n-1} \end{bmatrix} \tag{2.83}$$

where $p = [\kappa(x_{n-N'}, x_{n-N'+1}), \cdots, \kappa(x_{n-N'}, x_n)]^T$ and $k_{n-N',n-N'} = \kappa(x_{n-N'}, x_{n-N'})$. $\hat{\mathbf{K}}_n^{-1}$ can then be calculated using:

$$\hat{\mathbf{K}}_n^{-1} = \begin{bmatrix} \mathbf{K}_{n-1}^{-1}(\mathbf{I} + \mathbf{b}\mathbf{b}^T\mathbf{K}_{n-1}^{-1T}g) & -\mathbf{K}_{n-1}^{-1}\mathbf{b}g \\ -(\mathbf{K}_{n-1}^{-1}\mathbf{b})^T g & g \end{bmatrix} \tag{2.84}$$

where $\mathbf{b}$ is given by $k_n = [\mathbf{b}\ d]^T$ and g is given by $g = (d - \mathbf{b}^T\mathbf{K}_{n-1}^{-1}\mathbf{b})^{-1}$.

$\mathbf{K}^{-1}$ is then calculated using Equation (2.85)

$$\mathbf{K}_n^{-1} = \mathbf{G} - \mathbf{f}\mathbf{f}^T/e \tag{2.85}$$

where $\mathbf{G}$, e and $\mathbf{f}$ are given by:

$$\hat{\mathbf{K}}_n^{-1} = \begin{bmatrix} e & \mathbf{f}^T \\ \mathbf{f} & \mathbf{G} \end{bmatrix} . \tag{2.86}$$

and $\mathbf{G}$ is a $(N'-1) \times (N'-1)$ matrix, $\mathbf{f}$ is a $(N'-1) \times 1$ vector and e is a scalar.

Other pruning techniques have also been suggested including: approximate linear dependency (ALD) (Engel, Mannor, and Meir, 2004), the surprise criterion (Liu, Park, and Príncipe, 2009) and error minimisation (De Kruif and De Vries, 2003).

Since the SW-KRLS algorithm removes the oldest training pair with each new sample it is able to track non-stationary systems/processes. Psuedocode for the SW-KRLS algorithm, derived from (Van Vaerenbergh, Via, and Santamaria, 2006) and (Van Vaerenbergh, 2012) is provided in Algorithm 2.5.

Algorithm 2.5 Pseudocode: a training step for the SW-KRLS algorithm.

```
Initialise $\mathbf{K}_0$ as $(1+c)\mathbf{I}$ and $\mathbf{K}_0^{-1}$ as $\mathbf{I}/(1+c)$.
for $n = 1, 2, ...$ do
    Get $\hat{\mathbf{K}}_n$ from $\mathbf{K}_{n-1}$ with Equation (2.82)
    Calculate $\hat{\mathbf{K}}_{n-1}^{-1}$ from Equation (2.84)
    Get $\mathbf{K}_n$ from Equation (2.83)
    Calculate $\mathbf{K}_n^{-1}$ from Equation (2.85)
    Calculate $\alpha_n$ using $\alpha = \mathbf{K}_n^{-1}\mathbf{Y}_n$
end for
```

2.1.7 Support Vector Machines

SVMs, first described by Cortes and Vapnik (1995), are another use for kernel methods in machine learning. SVMs are designed for classification problems and regression problems. Some common classification problems include text and speech recognition. Least-squares support vector machines (LS-SVMs) solve the same least squares problem as the KRLS, however they are often solved using iterative techniques such as the sequential minimal optimisation (SMO) algorithm (Platt et al., 1998). Support vector machines are beyond the scope of this work, but, computationally the problem is very similar to kernelised least squares provided by the KRLS. As such, in Section 2.2 hardware implementations of SVMs are discussed and their architectures are analysed as this represents the most similar computational problem in the previous literature.

2.1.8 Summary

Section 2.1 touches on many aspects of machine learning and covers the fundamentals of kernel methods and KAFs. In this subsection, we summarise the key points and ideas that readers should take into the subsequent chapters. In particular, the readers should now be familiar with the following:

- kernel regression is an extension to linear regression to a (possibly) non-linear feature space through a Mercer kernel function, κ;
- the application of a kernel function to linear regression means that we can perform non-linear regression without directly computing the high-dimensional non-linear feature vectors;
- the Gaussian kernel is commonly used, because it is a universal approximator and it maps the input vectors to an infinite dimensional feature space;
- the Gaussian kernel can be intuitively thought of as the similarity between two input vectors;
- many KAFs are kernelised versions of linear adaptive filters, which have been expressed in the dual formulation to accommodate a Mercer kernel;
- KAFs are online algorithms which make small updates to their model with each new training example; and
- in order to apply any machine learning algorithm (including KAFs) to a problem, care must be taken to optimise the hyperparameters of the algorithm and to prevent overfitting, often using cross-validation.

2.2 High Performance Machine Learning

Machine learning techniques can be applied to a wide variety of applications. As such, performance problems in machine learning can be application specific. In applications where the processing of large amounts of data is required, memory bottlenecks may cause the majority of performance problems (Hsu et al., 2011). In other applications where the memory requirements are small, the latency may be critical and the input/output (I/O) latency may cause the majority of performance problems. In this section, performance problems in common machine learning applications are discussed with a particular focus on latency critical applications where the inputs may be the result of some high frequency sampled data. Previous hardware architectures for kernel based learning problems are also summarised.

2.2.1 High Throughput Machine Learning

High throughput machine learning is critical to the accessibility and performance of learning problems as often machine learning is performed on huge datasets. The throughput on a machine learning algorithm is often highly dependent on the application. In this subsection, several applications are presented and summarised in order to highlight several performance issues which may be encountered when attempting to develop high throughput machine learning systems. In many cases, machine learning problems require huge amounts of data and the problem of transferring and accessing this data becomes more important than the computational performance of the algorithm. For example, in the work by Hsu et al. (2011) bandwidth limitations are highlighted by an SGD algorithm running on a classification problem with 781K training examples and 60M of total non-unique features, using the dataset from Bottou (2008) which was derived from the work from Lewis et al. (2004). According to Hsu et al. (2011), SGD would take about 20 seconds to load this data and 0.4 seconds to learn a predictor. This shows that in some cases, memory bottlenecks outweigh the algorithmic bottlenecks. However, Hsu et al. (2011) notes that a performance of 0.4 seconds on such data represents performance roughly 100$\times$ slower than the peak performance of modern CPUs. This suggests that memory issues, such as cache misses, potentially play a role in machine learning problems.

In other problems, the complexity of the algorithm plays a larger role. For example, larger SVM classification problems, about 6000 features and 4000 training examples, GPU implementations (Athanasopoulos et al., 2011) can outperform an optimised CPU implementation, LIBSVM (Chang and Lin, 2011), by up to 10$\times$ as shown in Figure 2.6.

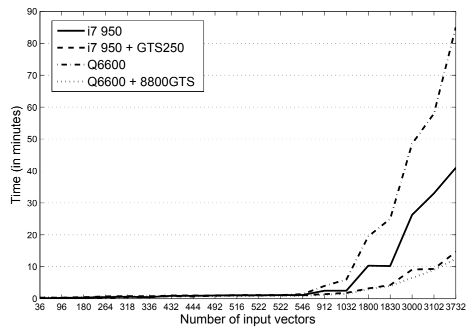


FIGURE 2.6: SVM training performance comparison - GPU SVM verses LIBSVM. Adapted from Athanasopoulos et al. (2011).

Machine learning problems can be much larger than those described by Athanasopoulos et al. (2011). In data mining problems, training examples can be as large as 10^6. For such datasets, when using a Nvidia GeForce 8800 GTX GPU, Do, Nguyen, and Poulet (2008) report a speedup of 100$\times$ against their own optimised CPU implementation for a LS-SVM problem and up to 1000$\times$ speedup over LIBSVM, both CPU implementations were run on an Intel Core 2 at 2.6GHz.

Lin, Lebedev, and Wawrzynek (2010) developed an FPGA based Bayesian computing machine capable of executing many algorithms which can be represented as Bayesian probabilistic networks. This includes several machine intelligence and signal processing problems. Lin, Lebedev, and Wawrzynek (2010) developed a sophisticated scheduling scheme in order to avoid memory stalls and their highly parallel FPGA implementation achieved on average 40$\times$/15$\times$ higher throughput than CPU/GPU implementations.

For very large machine learning problems it is common to distribute the problem across N machines in order to achieve up to $N\times$ speedup. Common methods for achieving this include feature sharding and instance sharding. Instance sharding involves splitting the set of training examples into N smaller training sets. These sets are then distributed to N computers. Each computer then uses a machine learning algorithm, like SGD, in order to develop a model for the training data. These N

models are then combined, often by taking the mean of all the parameters of each model, to create a new model. While this approach can be effective for machine learning problems with large datasets, the latency introduced by such an approach can increase by an order of magnitude (Hsu et al., 2011).

2.2.2 Low Latency Machine Learning

When machine learning algorithms are applied to digital signal processing (DSP) problems, often there are strict requirements on throughput and latency. For example, if a machine learning algorithm needs to be applied to some uniformly sampled data at a sample rate of F. Then in most cases the algorithm needs to provide new output at the same rate. For high sample rates, this high throughput is often achieved on CPUs by buffering the data. This can help one achieve high throughput but at the cost increasing the latency of the system. In other applications, such as algorithmic trading, the input data is not uniformly sampled but to be able to process the data with low latency provides an advantage as opportunities for "risk free" profitable trades may be possible.

2.2.3 FPGA Based SVM Implementations

SVM algorithms are usually quite complex with large datasets. As such, many challenges need to be overcome in order to produce highly optimised FPGA implementations. SVM algorithms also have large data dependencies within a single training step of the algorithm. This can be an obstruction for achieving low latency for kernel based algorithms but it does provide opportunities to develop intricate architectures in order to circumvent these dependencies. In this section, selected previous SVM architectures are summarised.

An early SVM architecture (Anguita, Boni, and Ridella, 2003) used 7 separate computing modules in order to train an SVM. The main SVM training block uses parallel processing elements in order to accelerate SVM computation.

Cadambi et al. (2009) use an FPGA co-processor in order to accelerate the dot product computations involved in SVM classification. The FPGA architecture is designed around a single instruction multiple data (SIMD) vector processor. A block diagram of the FPGA architecture is given in Figure 2.7.

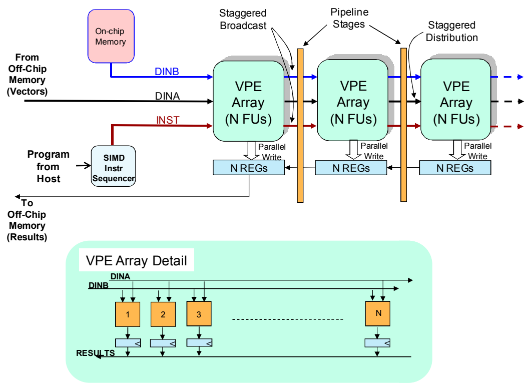


FIGURE 2.7: FPGA based co-processor for SVM classification. Adapted from Cadambi et al. (2009).

The heterogeneous CPU/FPGA architecture provides up to 21× speedup compared to CPU based SVM implementations. Use of a SIMD vector processor utilises the FPGA area more efficiently as processing elements are easily reused for different computations. This implementation is heterogeneous and as such suffers from latency from the Peripheral Component Interconnect Extended (PCI-X) bus and the operating system (OS) of the host machine.

2.2.4 Real-time Applications

In this section, we consider real-time applications where high performant implementations of KAFs may provide some additional benefit. For most of the applications described below, little study has been conducted on whether KAFs provide additional modelling accuracy over their linear counterparts. However, we expect that if there were a demonstration that KAFs could be implemented at sufficient data-rates for the applications below, that research in these subjects would become significantly more likely.

In the subsection, we briefly describe several real-time applications where KAFs may be applied.

Audio Compression

Many lossless audio compression algorithms contain linear prediction algorithms at their core. A channel of audio data can be thought of as a time series, x. The prediction algorithm operates over a window of previous examples up to x_n and attempts to predict the next sample, x_{n+1}. Let the prediction of the n^{th} sample be $\tilde{x}_n$. The prediction error, e, at sample n is given by $e_n = x_n - \tilde{x}_n$. For real audio signals, where samples are not necessarily independent, e is likely to have lower variance than the signal original signal, x. As such, e can potentially be stored at a lower bitrate than x using compression techniques such as Golomb-Rice coding (Golomb, 1966; Rice, 1979).

The general prediction equation is given by:

$$e_n = x_n - Q(\sum_{k=1}^{M} \hat{a}_k x_{n-k} - \sum_{k=1}^{N} \hat{b}_k e_{n-k}) \ , \tag{2.87}$$

where the function $Q(x)$ represents quantisation to the same wordlength as the original signal and $\hat{a}$,$\hat{b}$ represent the coefficients of the feedforward/feedback transfer functions given by $\hat{A}(z)$, $\hat{B}(z)$ respectively. A generic structure for a linear prediction equation is given in Figure 2.8.

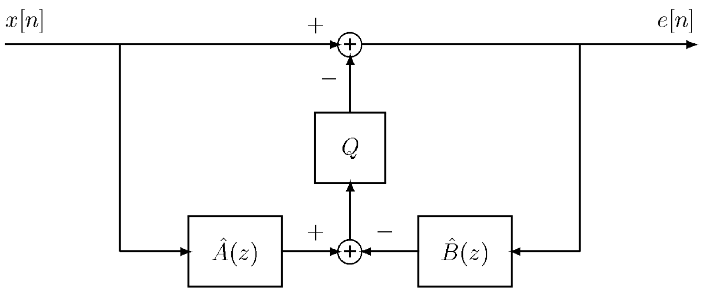


FIGURE 2.8: Linear predictor structure for encoding data. Adapted from Hans and Schafer (2001).

If $\hat{B}(z) = 0$ then the predictor behaves like a finite impulse response (FIR) filter, if $\hat{B}(z) \neq 0$ the predictor behaves like an infinite impulse response (IIR) filter. For reconstruction, Equation (2.87) can be rearranged and solved for x_n. The structure for reconstruction is shown in Figure 2.9.

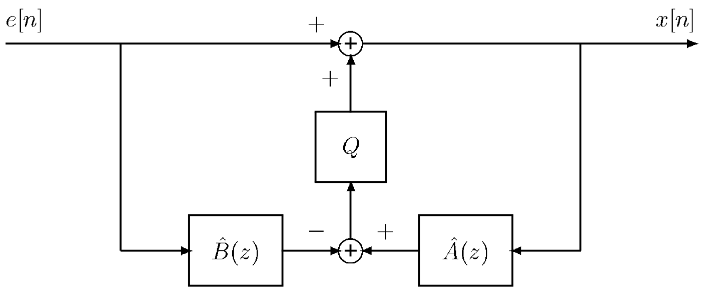


FIGURE 2.9: Linear predictor structure for reconstructing data. Adapted from Hans and Schafer (2001).

The coefficients $\hat{a}_n$ and $\hat{b}_n$ are often determined using an online tracking machine learning algorithm such the LMS or RLS algorithms. In order to provide a non-linear modelling to the predictor, occasionally polynomial features are added to the regression algorithm in order to improve its prediction capability.

Modern audio processing techniques, particularly mastering, involve many non-linear techniques such as dynamic range compression/expansion, limiting and clipping (Nielsen and Lund, 2000). Also, many sound sources in modern audio are better modelled as non-linear systems. For example guitar amplification, a common feature in popular music, behaves highly non-linearly (Pakarinen and Yeh, 2009). We propose that a general non-linear approximator, such as the KRLS algorithm with the Gaussian kernel, may provide higher audio compression than current lossless techniques. Finally, although in this section the focus was audio compression, these techniques are used in many types of signal compression and as such, the same algorithms may be used in a wide array of applications with significantly differing requirements.

Dereverberation

Dereverberation is the process of removing the effects of reverberation from signals. Dereverberation is a blind problem with no known solution (Naylor and Gaubitch, 2005). Given a sound source, $s(n)$, in an acoustic environment characterised by the impulse response, h, the dereverberation problem is characterised by

$$x(n) = h^T s(n) + v(n) \ , \tag{2.88}$$

where $x(n)$ is the reverberant signal and $v(n)$ is some system noise. The aim of a dereverberation algorithm is to extract the signal, $s(n)$. The effect of dereververation on speech is to improve its intelligibility. Figure 2.10 shows how the performance of speech recognition software improves with the reduction of reverberation (Kinoshita, Nakatani, and Miyoshi, 2006).

Dereverberation is often considered as a linear prediction problem which can be computed on the time series, x, using a linear regression algorithm. The linear prediction problem is often framed as follows

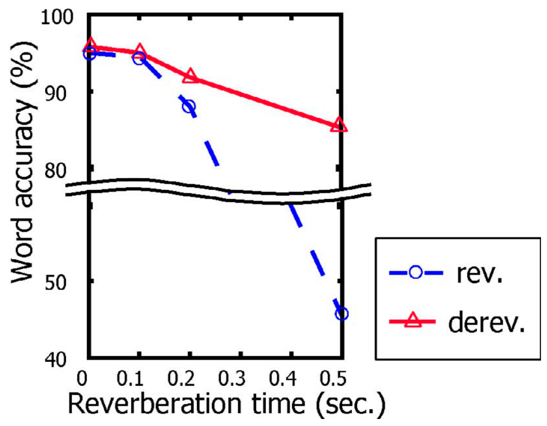


FIGURE 2.10: Speech recognition performance compared with different reverberation times. Adapted from Kinoshita, Nakatani, and Miyoshi (2006)

$$x(n) = \sum_{i=1}^{M} \alpha(i)x(n - i - D) + e(n) \ , \tag{2.89}$$

where α is a calculated set of coefficients, M is the filter order, $e(n)$ is the prediction error and D is the prediction delay. The error signal $e(n)$ is then used as the output of the dereverberation algorithm. The prediction delay D is introduced to prevent the dereverberation algorithm from removing parts of the signal that are wanted. For example, imagine the sound source, s, is an acoustic guitar and a concert hall. Sound is created by the guitar when an impulse is applied to a string, when a string is picked. After the initial picking, the string continues to vibrate in a feedback system due to the tension on the string and the structure of the guitar. Since this feedback system could be approximated using a linear model, its behaviour could be predicted by the linear predictor. However, the goal of dereverberation is not to remove the vibration of the string but to remove the effect the environment has on the sound of that vibrating string. The prediction delay (or "predelay"), D, is often adjusted experimentally depending on the sound source and the environment (Kinoshita, Nakatani, and Miyoshi, 2006). This prevents an algorithm with a large prediction delay from removing early reflections, however late reflections are often more important in improving tasks like speech recognition (Naylor and Gaubitch, 2005). Kinoshita, Nakatani, and Miyoshi (2006) also advocate adding white noise to the input signal, $x(n)$ in order to improve the calculation of the coefficients, α. Figure 2.11 shows the performance of the algorithm described by Kinoshita, Nakatani, and Miyoshi (2006) for an artificial dereverberation test.

Channel Equalisation

Lastly, let us consider the application of channel equalisation. Channel equalisation is the process of reconstructing a signal that has been corrupted as it passes through a communication channel, e.g., transmission along an electrical wire. Often, the distortion which corrupts the signal as it passes through the channel has time-varying characteristics. As such, a model to reconstruct the original signal from the corrupted one must be learned online and must be able to adapt to changes in the communication channel's characteristics. A common method to address this is with adaptive

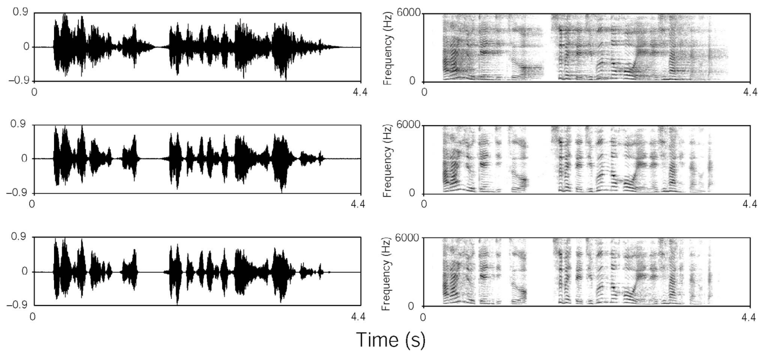


FIGURE 2.11: Waveforms of reverberant speech (top), original sound source (middle) and reconstruction from reverberant speech (bottom). Adapted from Kinoshita, Nakatani, and Miyoshi (2006)

filtering (Haykin, 2005). Commonly, the problem of channel equalisation is addressed as follows (Engel, Mannor, and Meir, 2004; Sebald and Bucklew, 2000):

1. a binary signal known to the receiver is transferred across the channel by the source;
2. the receiver receives the corrupted signal and creates a model to reconstruct the original binary signal;
3. the source now sends the actual binary data it wishes to transmit over the communication channel;
4. the receiver reconstructs the binary signal by quantising the result of it's prediction to either a '0' or a '1';
5. the receiver updates its reconstruction model to reduce the quantisation error between the real valued result at the output of it's model, and the quantised binary valued prediction.

All of the steps above except for the final one fall under the category of a generic online time series prediction problem. Note, that since the output is binary this is a classification task, but this can be addressed with regression algorithms followed by a simple quantiser. For example, one could map the binary values to either a -1 or a 1 and attempt to model these points using a regression algorithm, followed by the sign function.

Explicitly, given a binary signal: $\mathbf{u} = [u_0, \cdots, u_N]$, where $u_n \in \{-1, 1\} \forall n$ and some corrupted, measured signal: $\mathbf{y} = [y_0, \cdots, y_N]$, where $y_n \in \mathbb{R} \forall n$. The aim of channel equalisation is to reconstruct u_n using only samples from the measured signal, $[y_{n-1}, \cdots, y_{n-M}]$, where M is the feature length. At first, this task seems impossible, as we cannot make any assumptions about the desired signal, $\mathbf{u}$. However, there is some portion, let us say $\hat{N}$ samples, at the beginning of the signal $\mathbf{u}$ which is known to the receiver beforehand. We can call this mode the training mode of channel equalisation, which is followed by the online mode. As such, the desired signal for the entirety of training becomes:

$$d_n = \begin{cases} u_n & \text{if} n < \hat{N} \\ \operatorname{sign}\left(\tilde{u_n}\right) & \text{if} n \geq \hat{N} \end{cases}, \tag{2.90}$$

where d_n is the desired signal, $\tilde{u_n}$ is the a-prior prediction of u_n by the adaptive filter and $\mathrm{sign}(\cdot)$ is the sign function:

$$\mathrm{sign}\,(x) = \begin{cases} -1 & \mathrm{if} x < 0 \\ 1 & \mathrm{if} x \geq 0 \end{cases} . \tag{2.91}$$

Channel equalisation has been studied as a possible application for kernel methods and they show promising accuracy (Engel, Mannor, and Meir, 2004; Sebald and Bucklew, 2000). However, as far as we are aware there are no publications (excluding this work, Chapter 5) showing that kernel methods could be computed at rates fast enough for many channel equalisation problems.

2.3 Error Analysis

Computer arithmetic lies at the heart of any scientific computation. However, since most CPUs only support a few discrete different number representations in hardware, including: 32/64 bit integer/fixed point and 32/64 bit floating point, often only a few representations are considered when implementing algorithms in software. This is because the computational cost of using a number representation which is not supported in hardware will be large. FPGAs are reconfigurable by nature, and as such any bit depth/number representation could be chosen to implement a particular algorithm. Reducing the bit depth of an algorithm on FPGAs has the effect of reducing the area requirements and the algorithm while also increasing the maximum theoretical clock rate. The downside of reducing the bit depth is that, in general, round off error can cause an increase in error in the results or can cause an algorithm to have seemingly random output. The use of fixed point arithmetic, as opposed to floating point, is another means to reduce area requirements and increase the maximum theoretical clock rate. However, the range requirements of some machine learning algorithms mean that they may not benefit from the use of fixed point and care must be taken during the implementation of such algorithms. In this section, a background of error analysis in computer arithmetic is given along with some information on tools for automatic error analysis with a focus on MCA, an automatic stochastic method for error and sensitivity analysis.

2.3.1 Fixed Point

Fixed point numbers are a digital representation of numbers which are similar to integers. Like integers, fixed point numbers can be stored as unsigned or signed. Signed representations can take many forms including one's complement and commonly two's complement. The difference between an integer representation and a fixed point representation is the range of values that can be stored. An integer or fixed point number with a word length of B bits can represent 2^B unique numbers. For an integer representation in two's complement this can represent any integer in the range $[-2^{B-1}, 2^{B-1} - 1]$. For a fixed point representation in two's complement the range could potentially be bounded by any real number though there still are only 2^B representable numbers. In digital signal processing (DSP) a fixed point number usually has the range $[-1, (1 - 2^{1-B})]$. Where this fixed point representation is able to represent numbers throughout this range with a 2^{1-B} interval. Fixed point representations have a fraction length which is used to interpret a fixed point number into its true value. A fixed point representation using a fraction length of f and a word length of N can be denoted by $A(i, f)$. Where $i = N - f$ and sometimes referred to as the magnitude or integer bits. The quantisation of an infinite precision number, x, into a two's complement fixed point representation of $A(i, f)$ is given by (Yates, 2001):

$$Q_{i.f}(x) = -b_{B-1}2^{i-1} + \sum_{j=0}^{B-2} b_j 2^{j-f} \ , \tag{2.92}$$

where $Q_{i.f}$ is a function which maps x to the fixed point representation and b_n is the n^{th} bit of the resultant fixed point value. Usually, the bits $b_n \forall n$ are chosen such that $|x - Q_{i.f}(x)|$ is minimised (Lindstrom, Dahl, and Claesson, 2003) for some input x.

Rules of Fixed-Point Arithmetic

The following rules for fixed point arithmetic assume a two's complement representation of $A(a, b)$. The wordlength, w, of a fixed point number using this representation is given by:

$$w = a + b \ . \tag{2.93}$$

The range is given by:

$$-2^{a-1} \leq \alpha \leq 2^{a-1} - 2^{-b} \ . \tag{2.94}$$

During addition between two fixed point numbers, represented by $A(a_1, b_1)$ and $A(a_2, b_2)$, the output can be represented exactly by $A(\max(a_1, a_2) + 1, \max(b_1, b_2))$. If two fixed-point numbers, represented by $A(a_1, b_1)$ and $A(a_2, b_2)$, are multiplied together the result will be $A(a_1 + a_2, b_1 + b_2)$. In DSP algorithms, it is common that $a = 1$ and that the result of an operation will be rounded back to the same representation as the inputs. This results in a loss in precision which causes potential rounding error in the output of all fixed point operations.

2.3.2 Floating Point

Floating point number representation is conceptually similar to scientific notation used in science and mathematics. A generic floating point representation is given by:

$$\text{fl} = (-1)^s \cdot m \cdot \beta^e \ , \tag{2.95}$$

where s is the sign, m is the mantissa, e is the exponent and β is the base. In this section, it is assumed that $\beta = 2$ as this is the most common base used in modern computers.

We define $\circ$ as a generic floating point operation where $\circ$ is one of the following $\oplus \ominus \otimes \oslash$. These are the floating point equivalent of the exact operations $+ - \times /$. An assumption is made that for any floating-point operation $\circ$, the following holds:

$$(x \ \circ \ y) = (x \ \cdot \ y)(1 + \delta) \ , \tag{2.96}$$

where $|\delta| \leq u$, where u is the unit round-off of the precision which is being used and $(\cdot)$ represents an exact operation. The unit round-off is defined by:

$$u = \frac{1}{2}\beta^{1-t} \ . \tag{2.97}$$

This unit round-off assumes that the values x and y are able to be stored perfectly in the number system that we are using. The unit round-off for single precision is $u =$

$2^{-24} \approx 5.96\times10^{-8}$ and for double precision is $u = 2^{-53} \approx 1.11\times10^{-16}$ (Castaldo, 2007). These values for unit round off suggest that approximately $7/16$ significant decimal digits can be represented by single/double precision floating point respectively.

The error for a single floating-point operation is quite small and is usually accurate enough for most purposes. Unless catastrophic cancellation occurs when two very similar numbers are subtracted. In many modern scientific computations, the number of calculations that are performed can be enormous. As such, small errors will grow exponentially until they may cause large errors in the output. An example is when a vector is summed, as would occur in a dot product (Castaldo, 2007).

2.3.3 Error Analysis Techniques

Floating point error analysis can be grouped into two distinct techniques; static and dynamic error analysis. Static error analysis, such as the work by Goubault (2001), does not require code execution, rather a program is abstracted using mathematical techniques along with the semantics of the programming language. Often, static techniques create finite state machines (FSMs) or control flow graphs (CFGs) which completely define the mathematical behaviour of an algorithm which can then be analysed for robustness. Static error analysis techniques often do not scale well to large problems and often give overly cautious precision requirements, especially if those requirements are largely based on nature of the input data.

Dynamic error analysis, such as interval arithmetic (IA) (Hickey, Ju, and Van Emden, 2001), affine arithmetic (AA) (De Figueiredo and Stolfi, 2004), the contrôle et estemation stochastique des arrondis calculs (CESTAC) method (Vignes, 1996) and Monte Carlo arithmetic (MCA) (Parker, 1997), provide error analysis based on code execution.

Interval arithmetic (IA) (Hickey, Ju, and Van Emden, 2001) is a method whereby for each floating point operation, the error in the output is bounded by an interval. This interval is then used to calculate the error bounds for subsequent operations. The result is that the output, x, of any algorithm will lie in the interval $[x_l, x_h]$ which is calculated using IA. The error bounds which are found using interval arithmetic provide worst case performance of an algorithm, given a set of inputs, but in practise it tends to produce bounds which are overly pessimistic (Wilkinson, 1971).

Affine arithmetic (AA) is similar to IA as it provides error bounds on the output of an algorithm given a set of inputs. AA (Fang, Chen, and Rutenbar, 2003) attempts to compensate for the pessimistic error bounds given by IA. It does this by tracking the source of errors for each variable within an algorithm. Since the source of errors are known, these errors can be subtracted under certain conditions allowing for tighter error bounds in the output variable. This can be illustrated by the following example, given two floating point intervals, $\bar{x}$ and $\bar{y}$, both with the range $[-1, 1]$. Let these two values be related such that $\bar{y} = -\bar{x}$ and we wish to calculate the range for $\bar{z} = \bar{y} + \bar{x}$. Using standard IA, the calculated range for z is $[-2, 2]$. Using affine arithmetic, it is recognised that the errors for x and y are correlated and can be cancelled out. As such, the calculated range for z using affine arithmetic is $[0, 0]$. The range calculated by affine arithmetic still defines worst case error bounds but they are marginally tighter than ones calculated using interval arithmetic. Affine arithmetic is a powerful technique but requires large modifications to source code in order to implement.

The CESTAC technique (Vignes, 1996) uses normally distributed random rounding for each floating point operation. Using this method, the algorithm can be executed several times each producing a potentially different result. After running several simulations, frequency histograms can be examined to determine how many significant bits are lost during execution due to rounding error. This technique also provides insight into the sensitivity of the algorithm due to rounding. The CESTAC technique has been criticised for using a normal distribution to model rounding errors which can result in overly optimistic claims for accuracy (Kahan, 1996).

2.3.4 Monte Carlo Arithmetic

MCA is another stochastic method to estimate the effect of rounding error in floating point arithmetic. MCA applies random perturbations to the inputs and outputs of floating point operations such that the value for δ in Equation (2.96) becomes a random variable as opposed to a strictly defined value for each operation. MCA uses an *inexact* function (Parker, 1997) defined as:

$$inexact(x, t, \xi) = x + 2^{e_x - t}\xi \ , \tag{2.98}$$

where e_x is the value of the exponent of the floating point value given by $x \neq 0$, t is a positive integer representing *virtual precision* and ξ is a uniformly distributed random variable in the range of $(-1/2, 1/2)$. The inexact function is then applied to an arbitrary floating point operation as follows:

$$\mathrm{fl}(x \circ y) = round(inexact(inexact(x) \circ inexact(y))) \ , \tag{2.99}$$

where x, y are non-zero floating point numbers. If all floating-point operations are replaced with their MCA equivalents, then the output of floating point operations act like random variables. Large programs can then be executed using MCA over multiple runs and virtual precisions and the outputs may be analysed statistically in order to give the algorithm designer an understanding of its sensitivity due to rounding. If the algorithm is behaving in a numerically stable way at a virtual precision of t then its outputs often act like normally distributed random variables. As t decreases, the relative standard deviation of the output increases and can be considered as a loss in significant bits (Frechtling and Leong, 2015).

We advocate the use of MCA in determining the precision requirements of machine learning algorithms because:

- It provides data for robust sensitivity analysis due to rounding.
- It is a dynamic technique, which means it can provide specific precision requirements for a specific machine learning problem.
- It is faster than most other techniques.
- It can be run with very little modifications to the source code.

2.3.5 Mixed Precision

Mixed precision is the concept of using different number representations for different parts of an algorithm. For example, sections of an algorithm which have been identified as being sensitive to round-off error may be executed using a higher precision while other parts of an algorithm may be executed using a lower precision. This concept introduces several possible ways of optimising algorithms for higher performance and lower memory usage. Mixed precision also provides opportunities to increase the performance of hardware implementations. Consider an algorithm where thorough sensitivity analysis has been performed and it is determined that $h\%$

arithmetic operations need to be at a precision of H. The rest of the $l = 100\% - h\%$ operations may be performed at a precision of L. Any variable which is only used in the $l\%$ operations may only need to be stored at a precision of L thus introducing a space saving of $H/L\times$ for each identified variable. For operations involving large matrices, this space saving provides opportunities for better cache usage on CPU/GPU architectures thus reducing time spent waiting to retrieve data from global memory. If $H = L \times C$ where C is a positive integer, then opportunities for performance improvement as mixed precision hardware architectures may be used or designed in order to improve the performance of a calculation. For example, many modern CPUs/GPUs are theoretically capable of $2\times$ higher throughput if 32-bit floating point is used as opposed to 64-bit (Baboulin et al., 2009). For a Texas Instruments (TI) TMS320C6748 DSP processor, peak single precision performance is $4\times$ greater than double precision performance. For the case of FPGAs, arithmetic units may be designed so that they are capable of performing operations at any custom precision. This finer granularity allows hardware designers to pick specific datatypes for several parts of an algorithm's compute graph. As a result, FPGAs are capable of gaining a significant advantage from using mixed precision, compared with other platforms.

Chapter 3

Properties of Kernel Adaptive Filters and Accelerators

In any real system, there will be one or more application requirements which must be met, or that need to be optimised. Consider the following example: an object detection system for autonomous driving. The requirements of this system may be:

- accuracy (the algorithm needs to be able to detect and classify objects of interest);
- power (not exceeding a given power envelope);
- throughput (the system might be connected to a camera with a specific frame-rate); and
- latency (the system may need to react quickly to detected objects).

Similarly, other applications will have different requirements. In general, these requirements will often dictate what algorithm and hardware combinations are possible deployment options. In order to be sure which algorithm and hardware combinations will meet the applications, one needs to implement a given algorithm on a given platform to see if it meets the requirements. Doing this however, could incur significant costs if multiple platform / algorithm combinations need to be tried before one can be found which matches the application requirements.

In this chapter, we perform some algorithm and hardware analysis in order to understand the costs and benefits of different algorithms and different hardware platforms. While this doesn't necessarily guarantee performance on given hardware, it can give us an upper bound on throughput of a given algorithm on a given platform. The sections in this chapter cover memory bandwidth and throughput of several devices, along with the arithmetic intensity and precision requirements of several KAFs. We then identify applications, algorithms which are amenable for acceleration while also identifying some promising acceleration platforms.

3.1 Accelerator Platforms

Choosing an accelerator platform for which to deploy a particular algorithm is no easy task. Several requirements and desirable attributes that need to be taken into consideration, include:

- throughput,
- power consumption,
- latency and

- system integration.

A requirement could be any of the following:

- a certain throughput to match the line rate of attached data converters, or network interfaces;
- meeting certain power constraints to avoid a rack server from overheating; and
- being able to integrate the system with various sensors, or certain interfaces.

Similarly, desirable attributes could be:

- reducing energy consumption to extend the life of a battery powering the device;
- decreasing latency to achieve an edge over the competition (for example, in foreign exchange prediction); and
- increasing throughput, simply to reduce some associated server cost (for example, in server hire).

Clearly, these are not exhaustive lists. The specific requirements and desirables will regularly depend entirely on the target application. In this section, various properties of particular CPUs, GPUs and FPGAs are compared to understand the strengths and weaknesses of each platform. From this data, we can make informed decisions about which platform to select for a given application. In particular, we look at the power consumption, external memory bandwidth, operations per second and on-chip memory size. Finally, we use the information in this section to make algorithmic and architectural choices in subsequent chapters.

Firstly, let us pick some devices from ones we have available, and some others that may be utilised through cloud services, such as Amazon Web Sevices (AWS). From ones locally available, we select the following:

- an Intel Xeon E5-2670 (*Intel Xeon Processor E5-2670*)
- an Intel i7-4500U (*Intel Core i7-4500U Processor*)
- a Xilinx XC7Z020 (Xilinx, 2019e)
- a Xilinx XCZU3EG (Xilinx, 2018c)
- a Xilinx XC7VX485 (Xilinx, 2018a)

From devices available through services, we select the following:

- an Nvidia GRID K520 (*NVIDIA GRID K520*)
- a Xilinx XCVU9P (Xilinx, 2019b)

Overall, the device features which affect their possible choice as a deployment platform are in Table 3.1. Unless otherwise specified, the power consumption is quoted as the device thermal design power (TDP). The XC7Z020 power consumption is derived from the reference manual of the Pynq-Z1 board (*PYNQ-Z1 Reference Manual*). For the XCZU3EG, XC7VX485 and XCVU9P devices, board power is used from the Ultra96 (Avnet, 2018), VC707 (Xilinx, 2019c) and Alveo U200 (Xilinx, 2019a)

TABLE 3.1: Summary table of the properties of several devices of interest

Device	Mfr.	Tech. Node (nm)	Ext. Mem. BW (GB/s)	On-chip Mem. (MB)	Int16 Perf. (GOps/s)	Float32 Perf. (GFLOPS)	Power Cons. (W)
Xeon E5-2670	Intel	32	51.2	20.0	-	211.2	115
i7-4500U	Intel	22	25.6	4.0	-	96.0	15
GRID K520	Nvidia	28	320	0.5	-	4577	225†
XC7Z020	Xilinx	28	12.8	0.61	242	-	2.6
XCZU3EG	Xilinx	16	25.6	0.95	558	-	24†
XC7VX485	Xilinx	28	12.8	4.0	3640	-	60†
XCVU9P	Xilinx	16	77	43.2	10602	-	225†

†Maximum board power used from a commercially available product.

respectively. The peak floating point performance for the Intel processors was calculated using the technique proposed by Dolbeau (2015). For the Xilinx devices, the peak performance was calculated as follows:

$$P = 2N_{DSP}F_{Max} \quad , \tag{3.1}$$

where N_{DSP} is the number of DSP48 slices available on the given device, F_{Max} is the maximum operating frequency of those DSPs. The factor 2 denotes that each DSP slice can calculate a multiply accumulate (MAC) every cycle. The F_{Max} of the XC7Z020, XCZU3EG, XC7VX485, XCVU9P is 551MHz (Xilinx, 2018b), 775MHz (Xilinx, 2019f), 650MHz (Xilinx, 2018a), 775MHz (Xilinx, 2019d) respectively. Note, for F_{Max}, we assume a -2 speed grade device. Clearly, each of the devices available are not necessarily equivalent. The most similar devices across manufacturers are the Xeon E5-2670, GRID K520 and XC7VX485 which are from a similar time period, rely on similar manufacturing technology and consume similar amount of power. The DSP slices available in the Xilinx devices work well for Int16 computations, while the Nvidia and Intel devices are optimised for single precision floating point compute. While, all devices are capable of floating point and Int16 compute, the Xilinx devices will perform roughly an order-of-magnitude slower than their quoted Int16 performance. Clearly to utilise an Int16 datatype, the numerical stability of the algorithms at hand must be amenable to performing correctly with a low-precision datatype. The Nvidia platform features significantly higher external memory bandwidth, meaning it will scale better for more memory demanding problems. The Xilinx and Intel platforms have significantly more on-chip memory than the Nvidia platform which could be used to achieve better performance for smaller problems where all the memory associated with an algorithm can fit purely on-chip.

3.1.1 Roofline Analysis

Table 3.1 shows several key pieces of information about each available device, but it's still difficult to distill this information into some key aspects which help us understand if a given device is suitable for a given application. In this section we introduce the roofline model (Williams, Waterman, and Patterson, 2009) associated with each device, which provides more insights into selecting a particular hardware platform over another.

Figure 3.1 shows the roofline model associated with each of our selected devices. The x-axis shows AI, i.e., the number of arithmetic operations performed per byte of memory read from memory. The y-axis shows performance in GOps/s. Each line represents a performance ceiling for each device depending on the AI of the particular algorithm which is being implemented. The sloped lines of the figure

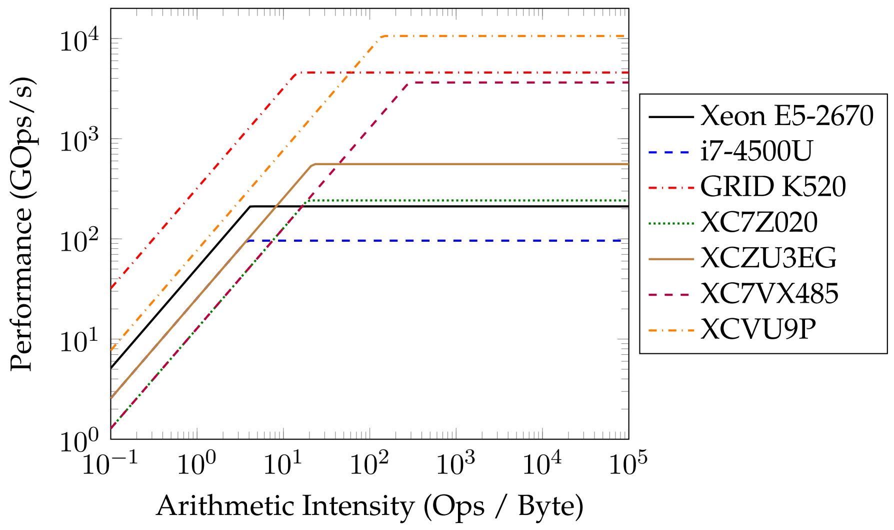


FIGURE 3.1: Rooflines of several different CPU, GPU and FPGA devices.

represent algorithms which are *memory bound* while the flat portions of the figure represent algorithms which are *performance bound*. The Nvidia device has very high external memory bandwidth, and a very high peak performance ceiling. The Intel and Xilinx devices have low external memory bandwidth, while the Xilinx devices have generally higher peak performance ceilings. Note, these performance ceilings represent Int16 for the Xilinx platforms, while they represent single precision floating point for the CPU and GPU platforms. Again, this means that the application algorithms themselves must be amenable to implementation with a 16-bit integer datatype, otherwise the CPU/GPU platforms will be much more attractive for such algorithms. At first glance, it may appear that the GRID K520 platform is the best platform on average. It provides very high bandwidth for low AI applications and quite a high performance ceiling for high AI applications. However, this type of roofline analysis can miss some of the subtleties associated with each device. For example, the peak performance ceiling itself does not capture the granularity of parallelism which is available on the target device, i.e., the GRID K520 provides high amount of course-grained parallelism which can easily be exploited for embarrassingly parallel problems, while the CPU and FPGA platforms can easily accelerate applications with a finer granularity of parallelism, such as those with internal dependencies. All of those subtleties are difficult to capture without providing an algorithm or implementation, but the performance ceiling provided by roofline analysis is useful in:

- determining the theoretical peak performance of a given algorithm (with a certain AI) on a given device;
- understanding whether certain algorithms are memory or compute bound on a particular device; and
- how close a given implementation of an algorithm is to the device's theoretical peak performance.

In the next section, we will look at how these device properties can be combined with algorithmic properties of KAFs to make informed decisions about choosing algorithms and acceleration platforms.

3.2 Algorithmic Analysis of Kernel Adaptive Filters

In this section, the properties of several KAFs are analysed under several different assumptions. The main metric that is of interest is *arithmetic intensity*, which is the number of mathematical operations per byte of memory read and written. The following algorithms are considered: KRLS (Engel, Mannor, and Meir, 2004), KNLMS (Richard, Bermudez, and Honeine, 2009) and quantised kernel least mean squares (QKLMS) (Chen et al., 2012). In all cases we're considering the training phase of the development. A calculation of arithmetic intensity is provided for small scale problems (where the predictive model fits in on-chip memory of the device) and large scale problems where the predictive model is stored in external memory. Throughout the analysis, we assume that scalar values within each algorithm can be stored in on-chip memory / registers and therefore are not included in the value for memory reads.

3.2.1 Notation

The following notation is used throughout this section:

- $\mathbf{x}_n$, the n^{th} input vector.
- y_n, the n^{th} output or target value.
- M, the length of the input vector.
- N, the size of the training set.
- $\mathcal{D}$, the dictionary/support vectors (a subset of input vectors).
- $\tilde{N}$, the size of the dictionary.
- $\tilde{x}_i$, the i^{th} entry in the dictionary.
- $\boldsymbol{\alpha}$, the vector of weights (one element for each dictionary entry).
- κ, the kernel function. Unless stated otherwise, the exponential kernel is assumed: $\kappa(\mathbf{x}_i, \mathbf{x}_j) = e^{-\gamma\|\mathbf{x}_i - \mathbf{x}_j\|_2^2}$. Note that with other kernels, the arithmetic intensity will be slightly different.
- w, the wordlength for the datatype (in bytes) which is in use.

For a recap on what these terms mean, see Section 2.1.

3.2.2 Compute and Memory Requirements

In this section, the numbers of operations / memory read are computed for several KAFs (Liu, Príncipe, and Haykin, 2011). Specifically, KRLS (Engel, Mannor, and Meir, 2004), KNLMS (Richard, Bermudez, and Honeine, 2009) and QKLMS (Chen et al., 2012). The implementation of QKLMS uses the coherence criterion (Richard, Bermudez, and Honeine, 2009) instead of Euclidean distance to select the dictionary entries, which saves computations for that particular algorithm. Note that while the following algorithms have different arithmetic intensities, they also have different performance in terms of their modelling accuracy / OP and accuracy / memory requirements. Van Vaerenbergh and Santamarıa (Van Vaerenbergh and Santamarıa,

TABLE 3.2: Summary of the operations required to update the model based on a new example

Algorithm	KRLS	KNLMS	QKLMS
exp	$\tilde{N}$	$\tilde{N}$	$\tilde{N}$
$\times$	$3\tilde{N}^2 + M\tilde{N} + 5\tilde{N} + \sum_{i=1}^{\tilde{N}} i$	$(M+4)\tilde{N} + 1$	$(M+2)\tilde{N} + 1$
$+$	$3\tilde{N}^2 + 3M\tilde{N} + \sum_{i=1}^{\tilde{N}} i$	$2(M+1)\tilde{N}$	$2M\tilde{N} + 1$
$\div$	1	1	-
$<$	1	$\tilde{N}$	$\tilde{N}$
Total	$6\tilde{N}^2 + 4M\tilde{N} + 6\tilde{N} + 2 + 2\sum_{i+1}^{\tilde{N}} i$	$(3M+8)\tilde{N} + 2$	$(3M+4)\tilde{N} + 2$
Mem. (input)	$(M+1)w$	$(M+1)w$	$(M+1)w$
Mem. (model)	$(\tilde{N}M + \tilde{N} + 2\sum_{i=1}^{\tilde{N}} i)w$	$(\tilde{N} + \tilde{N}M)w$	$(\tilde{N}M + \tilde{N})w$
Mem. (update)	$(\tilde{N} + \sum_{i=1}^{\tilde{N}} i)w$	$\tilde{N}w$	w
Intensity as $\tilde{N} \to \infty$	$4.7/w$	$(3M+8)/(M+2)w$	$(3M+4)/(M+1)w$
Intensity as $M \to \infty$	$4/(\tilde{N}+1)w$	$3\tilde{N}/(\tilde{N}+1)w$	$3\tilde{N}/(\tilde{N}+1)w$
Intensity as $\tilde{N}, M \to \infty$	$4.4/w$	$3/w$	$3/w$

2013) provide a nice comparison between the modelling accuracy, floating point operations and memory requirements of a number of popular KAFs. Their comparison could be used in conjunction with the static analysis provided here to determine which are the most promising KAFs suitable for implementation. In order to calculate an incremental update based on a new training example, $\{\mathbf{x}_n, y_n\}$, several operations are required. These are summarised in the Table 3.2. Operations are split up into the different common operators found in each algorithm. In practice, exponential evaluation may be performed using polynomial evaluation (several multiply and accumulates if Horner's method is used), but no such substitution appears in Table 3.2. Memory is split into three components:

1. memory reads required to receive the next input example;
2. memory reads required to access the model parameters; and
3. memory writes required to update the model parameters.

For both memory accesses and operations, we assume the current entry is not added to the dictionary. To calculate the arithmetic intensity for large models, we assume that both the input and the model parameters are stored off-chip.

Interestingly, Table 3.2 shows very low AI as $\tilde{N}, M \to \infty$, in the range of $3 \to 4.4$ operations per datatype word. For single precision floating point $w = 4$, and for Int16 $w = 2$, meaning the actual AI is $0.75 \to 1.1$ for single precision floating point and $1.5 \to 2.2$ for Int16. These AI numbers refer to how the algorithm scales as the size of the internal model increases. However, there are extra datapoints to consider. For example, for problems where the entire model fits in on-chip memory, the only external memory reads / writes will be:

- fetching the latest training pair, $\{\mathbf{x}_n, y_n\}$, from an external source (let us assume it's from external memory); and
- optionally writing back the a-prioi prediction, $\tilde{y}_n$, to an external sink (again, let us assume it's to external memory).

Furthermore for small time series prediction problems, the memory read operations per update reduces even further to just two words of data.

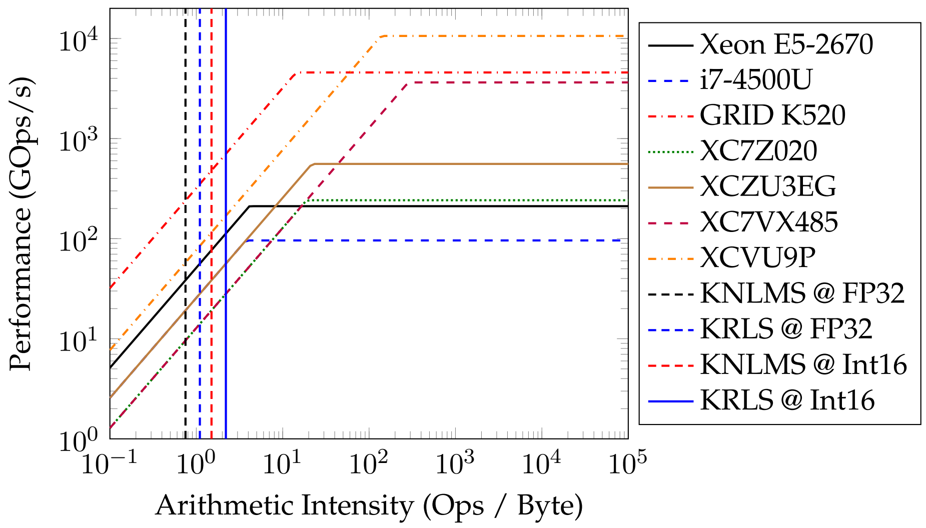


FIGURE 3.2: Rooflines of several different CPU, GPU and FPGA devices.

3.2.3 Hardware Implications

Putting together all of the information from Sections 3.1 and 3.2.2 we can build a picture for if / when a certain device should be used over a another one. Let us add vertical lines to our previous roofline figure, Figure 3.1, which denotes the AI of each algorithm under various assumptions. Figure 3.2 shows this updated figure.

The arithmetic intesity shown in Figure 3.2 uses the large model assumption, as $\tilde{N}, M \to \infty$. Recall that the performance ceilings for the FPGA platforms relate to Int16 computations, while the CPU and GPU platforms relate to single precision floating point. As a result, we have 4 vertical lines corresponding to two algorithms, KRLS and KNLMS,[1] with two different datatypes, single precision floating point and Int16. Clearly, with the large model assumption we can see that these algorithms are memory bound for all platforms of interest. With this in mind, it is clear that the GPU device is the best platform available to accelerate these algorithms with the large model assumption. Although this view ignores some of the subtleties of fine-grained versus coarse-grained parallelism, it is likely under the large model assumption the coarse-grained parallelism available on the GPU can be well utilised for large vectors and matrices. In terms of overall performance, an efficient implementation of these KAFs would be an order of magnitude below the peak performance of each device.

Secondly, let us consider the AI of the largest model which can fit on the on-chip memory of each device. In this scenario, we only consider the reading of the latest target, y_n, and the writing of the a-priori prediction, $\tilde{y}_n$ as the memory reads / writes respectively. We select values for $\tilde{N}$, M which produce the highest AI for each device, with the constraint that the model memory (from Table 3.2) fits in the device on-chip memory. As the memory reads are constant for each configuration, we can simply model this search as the following constrained optimisation problem:

$$\tilde{N}_{max}, M_{max} = \underset{\tilde{N}, M \in \mathbb{N}^*}{\operatorname{argmax}} \left(O_{alg}(\tilde{N}, M) \right) \;\; \text{s.t.} \;\; S_{alg}(\tilde{N}, M) w \leq S_{device} \;\; , \tag{3.2}$$

where $O_{alg}(\tilde{N}, M)$ is a function which returns the number of operations for a given algorithm, given values for $\tilde{N}$, M (i.e., the "Total" row in Table 3.2), $S_{alg}(\tilde{N}, M)$ is

[1]QKLMS is omitted as it has the same AI as KNLMS

TABLE 3.3: Maximum arithmetic intensity for models which fit on-chip for all algorithms and all hardware configurations

Algorithm		KRLS			KNLMS		
Device	w	$\tilde{N}_{max}$	M_{max}	AI (Ops / Byte)	$\tilde{N}_{max}$	M_{max}	AI (Ops / Byte)
Xeon E5-2670	4	2288	1	4 583 722.25	2621440	1	3 604 480.25
i7-4500U	4	1022	2	915 840.0	524288	1	720 896.25
GRID K520	4	360	2	114 075.25	65536	1	90 112.25
XC7Z020	2	510	2	457 088.0	131072	1	360 448.5
XCZU3EG	2	704	1	869 264.5	249036	1	684 849.5
XC7VX485	2	1446	2	3 664 526.0	1048576	1	2 883 584.5
XCVU9P	2	4757	1	39 618 675.0	11324620	1	31 142 705.5

a function which returns the size of the model for a given algorithm, given values for $\tilde{N}$, M (i.e., the "Mem. (model)" row in Table 3.2), and S_{device} is the amount of memory available on a given device.

The results of solving the constrained optimisation problem given in Equation (3.2) for each device and algorithm is given in Table 3.3. Clearly on average, increasing $\tilde{N}$ results in more operations per update than increasing M, as such, $M_{max} \in \{1, 2\}$ while $\tilde{N}_{max}$ is much larger, and varies depending on the algorithm and the memory capacity of the device. Note, QKLMS is omitted from this table, as it resulted in identical values for M_{max} and $\tilde{N}_{max}$ as KNLMS. Furthermore, the AI was negligibly different to KNLMS. The values for $\tilde{N}_{max}$ are significantly larger for KNLMS than for KRLS, this is because KRLS needs to store an inverse matrix, $\tilde{\mathbf{K}}^{-1} \in \mathbb{R}^{\tilde{N}_{max} \times \tilde{N}_{max}}$, in order to calculate the optimal model update for each new training example. KNLMS and QKLMS are SGD based algorithms and do not have this constraint. As expected, with the different storage model, i.e., all model memory stored on-chip, each configuration has extremely high AI, as such, all of these configurations are compute bound on all devices. When algorithms have high AIs, the performance of the compute kernels used to implement them become vitally important. Any pipeline bubble or processor stall will be noticed in the overall throughput of the hardware. It's this property, along with the larger on-chip memory, that makes FPGAs an attractive platform for implementing KAFs. The larger on-chip memory allows for larger models to be trained before needing to spill to external memory, and the flexibility in hardware allows one to design a custom accelerator for the given algorithm that can experience fewer pipeline bubbles and processor stalls than general purpose processors, like CPUs and GPUs.

3.3 Precision Requirements of Kernel Adaptive Filters

Another aspect of an algorithm which has significant impact on it's amenability to high performance implementations is the precision. Intel CPUs / Nvidia GPUs often have a $2\times$ / $10\times$ speedup when using single precision floating point over double precision floating point. For FPGAs it becomes even more interesting, as an arithmetic unit can be implemented either with DSP blocks, LUTs or a combination of both. This combination of DSPs and LUTs means that FPGAs are able to exploit reduced precision arithmetic at a much finer granularity than their CPU/GPU counterparts. Specifically, a reduction in the number of bits required results in less logic resources being used to implement that arithmetic, as such a higher computational density can be achieved on the device. FPGAs have been shown to be efficient at performing arithmetic at extremely low precision, e.g., 1-bit to 3-bit operands, which can provide significantly improved performance and lower overall memory footprint (Blott et al., 2018).

In this section, we look at the sensitivity of KAFs to rounding using MCA (Parker, 1997).

3.3.1 Precision Analysis of Kernel Adaptive Filters Using Monte Carlo Arithmetic

MCA involves adding noise to inputs and outputs of every floating point operation in an algorithm (Parker, 1997). Due to this, floating point numbers behave like random variables on which we can perform some statistical analysis. In this section, we use MCA as a tool to estimate the amenability of several KAFs to rounding. As such, we can understand which algorithms are more amenable to highly performant FPGA implementations. In order to do this, we leverage MCALIB (Frechtling and Leong, 2015) to apply MCA where regular floating point operations would occur. MCALIB introduces a *virtual precision*, t, which determines how many bits will be randomly modified at the input and output of each operation. The virtual precision, t, refers to the number of bits in the mantissa (including the implicit bit), while the number of exponent bits and the sign bit does not change. With this in mind, setting t to various values can approximate several popular floating point formats, including:

- double precision floating point, when $t = 53$;
- a mantissa size equivalent to single precision floating point when $t = 24$;
- a mantissa size equivalent to half precision floating point when $t = 11$; and
- a mantissa size equivalent to BFLOAT16 (Kalamkar et al., 2019) when $t = 8$.

Note that only the mantissa size in modelled for single precision floating point, half precision floating point and BFLOAT16, due to MCALIB using double precision floating point as its carrier datatype. With this in mind, care needs to be taken to ensure underflow or overflow would not occur in the modelled datatype. This is defined by the exponent size of single precision floating point, half precision floating point and BFLOAT16 being 8, 5 and 8 respectively, compared to the exponent size of double precision floating point, of 11. We then perform 100 trials of MCA at each value of t and observe the relative standard deviation in the output. The relative standard deviation tells us two things:

1. the amount of noise the virtual precision will introduce in the result due to rounding, i.e., the number of bits of precision that are *lost* during the computation;
2. the value for t for which the algorithm still behaves in a stable way, i.e., values for t which purely result in an increase in noise in the output, rather than numerical instability.

In order to test the sensitivity of KAFs to rounding, two algorithms are tested: SW-KRLS (Van Vaerenbergh, Via, and Santamaria, 2006) and KNLMS (Richard, Bermudez, and Honeine, 2009). SW-KRLS is a recursive least-squares algorithm which calculates an inverse matrix update using a few operations via the matrix inversion lemma (Woodbury, 1950). KNLMS is a stochastic gradient descent type algorithm with its update equations resembling LMS while also utilising a kernel function (Aronszajn, 1950). The inverse matrix calculation of SW-KRLS allows an

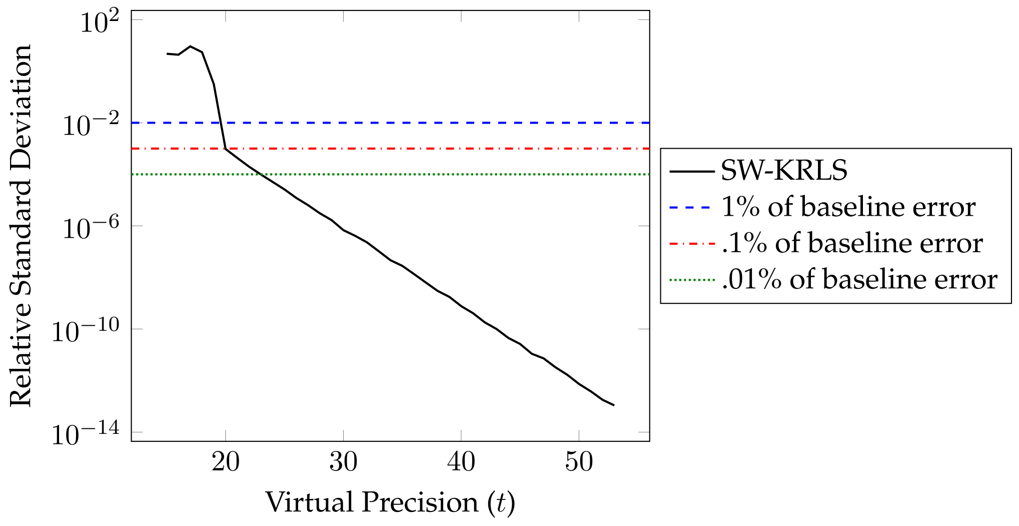


FIGURE 3.3: Sensitivity of SW-KRLS due to rounding.

optimal model to be calculated for every update, while KNLMS tends towards an optimal model over time. These two algorithms represent the two main types of KAFs. We expect other RLS based KAFs to behave similarly to SW-KRLS and other LMS based KAFs to behave similarly to KNLMS. For the input, the Mackey-Glass chaotic time series (Mackey, Glass, et al., 1977) was used. This time series is generated with the following differential equation:

$$dx(t)/dt = -ax(t) + bx(t-\tau)/(1 + x(t-\tau)^{10}) \quad , \tag{3.3}$$

where $a = 0.1$, $b = 0.2$ and $\tau = 30$. A feature length of 7 is used throughout, $M = 7$, i.e., the proceeding 7 samples are used to predict the next example. The training set contains 3000 samples, and the last 500 examples are used to evaluate each algorithm. MSE is used as the evaluation metric, where a lower value corresponds to better modelling accuracy. This configuration is identical to the configuration used by Engel, Mannor, and Meir (2004). We chose hyperparameters from Van Vaerenbergh (2012). Explicitly, for SW-KRLS we use: $\tilde{N} = 20$ and $c = 0.0001$, where $\tilde{N}$ is the sliding window length and c is the regularisation factor. For KNLMS we use: $\mu_0 = 0.9$, $\eta = 0.1$ and $\epsilon = 0.0001$, where μ_0 is the coherence criterion parameter, η is the step-size and ϵ is the regularisation factor. For both SW-KRLS and KNLMS we use the Gaussian kernel, given by: $\kappa(\mathbf{x},\mathbf{v}) = e^{-\frac{\|\mathbf{x}-\mathbf{v}\|^2}{2\sigma^2}}$, where $\sigma = 0.6$. When double precision floating point is used without MCA a MSE of $MSE_{SW-KRLS} = 7.38 \times 10^{-3}$ for SW-KRLS and $MSE_{KNLMS} = 9.47 \times 10^{-3}$ for KNLMS. These serve as baselines throughout this section. Figure 3.3 shows the relative error, represented here as the absolute relative change in MSE, which occurs as a result of using MCA at various virtual precisions. There are also 3 horizontal reference lines which signify when there is a 1%, 0.1% and 0.01% relative standard deviation. When these lines intersect with the measured relative standard deviation line for SW-KRLS, this tells us the t value associated with that amount of error tolerance. The straight line between $t = 20$ and $t = 53$ is the region where the SW-KRLS algorithm is behaving as expected. The values of t within this range signify various levels of noise due to rounding. In the region where $t < 20$, the algorithm is not longer behaving as expected. It's likely that some catastrophic cancellation (Higham, 1996) is occurring during the computations at these values of t. In this range small changes to the input will

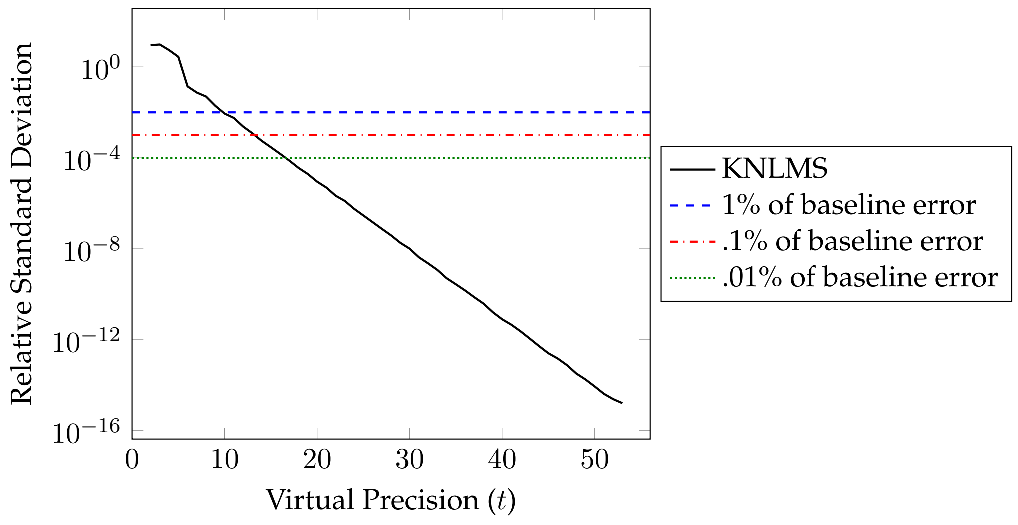


FIGURE 3.4: Sensitivity of KNLMS due to rounding.

result in high variance in the modelling capability of SW-KRLS. Considering the 3 reference lines, it's possible that an application requirement would tolerate any of the suggested relative increases in MSE. However, the 1% bar, $t = 19$, falls into the region of algorithmic instability and as such, would not be recommended. The 0.1% bar, $t = 20$, is right on the edge of algorithmic instability and again is probably not a recommended value for t. The 0.01% bar, $t = 24$, is in the stable region with a few bits of padding, and corresponds to single precision floating point. Given the popularity and availability of the this format, single precision floating point would be a good candidate datatype for SW-KRLS, assuming that this level of relative increase in MSE was acceptable. Finally, note that this assumes that all values within the computation fall within the exponent range of 8-bits, which is highly likely.

Figure 3.4 shows the relative error which occurs as a result of using MCA at various virtual precisions. Again, the relative error refers to absolute relative change in MSE. There are also the same 3 horizontal reference lines which signify when there is a 1%, 0.1% and 0.01% relative standard deviation. Again, when these lines intersect with the measured relative standard deviation line for KNLMS, this will tell us the t value associated with that amount of error tolerance. KNLMS appears to be significantly more stable than SW-KRLS. For example, the stable region from KNLMS is from $t = 6$ to $t = 53$. This is 14 bits lower than the lower bound for SW-KRLS. Below that value, $t < 6$, KNLMS begins to behave in an unstable way, meaning that datatypes below this precision will behave unpredictably for KNLMS. For KNLMS all 3 reference lines, fall in the region of algorithmic stability. This suggests that all three reference points may be useful for applications which tolerate these levels of relative increase in MSE. The 1%, 0.1% and 0.01% occur at virtual precisions $t = 10$, $t = 13$ and $t = 17$ respectively. On that basis of these results, it appears a half precision floating point format or some other kind of custom floating point format would work well with KNLMS. Note, this also assumes that the exponent does not overflow or underflow. Even so, it is likely that simple bias shifting (known in the deep learning literature as *loss scaling* (Micikevicius et al., 2017)) would alleviate the problem.

Given the results, KNLMS can be considered as a suitable candidate for high performance implementations. Particularly, on FPGA platforms where custom datatypes can be tailored to the application requirements and can significantly increase their computational density. For both SW-KRLS and KNLMS, MCA only tested the mantissa sizes of floating point datatypes. Although a similar trend for fixed point datatypes is likely, it is also likely that several extra bits may be required to compensate for the dynamic range.

3.4 Conclusion

In this chapter, we looked at potential accelerator platforms, the arithmetic intensity of several KAFs and the amenability of those KAFs to quantisation. Based on the large model roofline assumption, it was shown that GPUs would be superior at accelerating large scale KAFs. This is due to KAFs being memory bound as the size of the model size increases, as such, GPUs excel due to their higher external memory bandwidth. CPUs and FPGAs were seen as good candidates for smaller scale problems, due to their larger on-chip memory storage. For high performance implementations, SGD based algorithms like KNLMS seem to be good candidates, as their on-chip memory requirements are lower per dictionary entry. Furthermore, KNLMS appears more amenable to heavy quantisation and as such, may be able to exploit the potential higher compute density available on FPGAs at lower bit widths.

The next two chapters in this document, take the lessons learned from this chapter and as a result, two different core generators are created for KNLMS based KAFs targeting FPGAs.

Chapter 4

FPGA Accelerators for Hyperparameter Optimisation

4.1 Introduction

Machine learning and data mining focus on the development of mathematical ideas and algorithms to learn from data. Interest in these fields has been steadily increasing in recent years as advancements have addressed previously intractable problems such as speech recognition, handwriting recognition, image processing, credit card fraud and automatic fault detection. Kernel methods are an important class of machine learning algorithms which include support vector machines (SVMs) (Scholkopf and Smola, 2001), Gaussian processes (GPs) (Rasmussen and Williams, 2006), and kernel adaptive filters (KAFs) (Liu, Príncipe, and Haykin, 2011).

Reconfigurable computing, the application of field programmable gate arrays (FPGAs) to computing problems, has been successfully applied in accelerating certain classes of problems. The following computational conditions are desirable for an efficient FPGA implementation: (1) instruction and task level parallelism; (2) high ratio of computation to memory accesses (arithmetic intensity); (3) modest precision requirements; and (4) low input/output bandwidth.

Standard implementations of SVMs (Platt et al., 1998) and GPs (Lawrence, Seeger, and Herbrich, 2003) involve *batch-mode* algorithms which perform multiple passes over the training set in order to converge towards an optimal solution/model. These have time complexities which are $O(n_{TST}^2)$ or higher, where n_{TST} is the number of training samples, and the resultant models cannot be incrementally updated when new data becomes available. They do not satisfy conditions (1)-(2) because: storage of the entire training set is required; the result of one iteration is required before the next iteration can proceed; and many memory accesses are required per data input with processing time increasing with data size. In contrast, KAFs are recursive algorithms which perform a small, fixed amount of computation per data input, and meet all the above conditions, making them amenable to efficient FPGA implementations.

Different KAF algorithms have been proposed for classification, regression and anomaly detection tasks (Liu, Príncipe, and Haykin, 2011). In this chapter, we describe a particularly efficient implementation of the kernel normalised least mean squares (KNLMS) algorithm. KNLMS was chosen because of its simple computational structure and its ability to approximate any continuous function with arbitrary accuracy (Liu, Príncipe, and Haykin, 2011). The computational bottleneck for KNLMS is the evaluation of an inner product in the feature space. Our implementation is heavily pipelined which leads to high latency which is normally undesirable. However, our core is specifically designed to address the problem of algorithm configuration in machine learning, which is known as a *hyperparameter search* (Bergstra and Bengio, 2012a). For a given machine learning algorithm with P parameters, each parameter needs to be tuned to suit the dataset at hand. If P parameters are explored, at B different values, the number of hyperparameter sets is $n_h = B^P$. Training needs to be performed on each hyperparameter set to determine its suitability to the given dataset. We exploit the independence of the parameter search to evaluate L independent parameter settings in parallel, neatly filling the KNLMS pipeline with L cycles of latency. As a result, our implementation achieves very high computational efficiency.

The key contributions of this work are:

- The first fully pipelined datapath for a KAF. Compared with previous vector-processor architectures, much higher performance can be attained because all pipeline stages do useful work and never stall. Pipeline latency is addressed by filling all stages with independent parts of a hyperparameter search.
- A number of optimisations for the KNLMS algorithm: pipelining, memory optimisations, and scheduling are combined to achieve a $575\times$ speedup over a naïve implementation for parameter optimisation for floating point arithmetic.
- A complete PCI Express (PCIe)-based system implementation with a speedup of $10\times$ over a processor, $2.66\times$ over a GPU and a speedup of $660\times$ over a previous microcoded kernel recursive least squares implementation, by Pang et al. (2013).

Contributions of this chapter include: expanded background and architecture sections; a new fixed-point implementation which uses 35% of LUTs and 17% of DSPs, while achieving 60% lower latency than the previously reported floating-point implementation; addition of a random search which often finds better hyperparameters than the previously reported grid search (Bergstra and Bengio, 2012a); and an improved PCIe interface which obviates the transfer of n_h regression values per input vector and achieves $4\times$ higher throughput over the previously reported design. In this work, we've also included the results of a much improved CPU implementation of a parameter search and also included a GPU implementation as a comparison. Finally, different problem sizes have been considered by experimenting with folding the algorithm on the core, allowing larger problems to be tackled while reducing the cores throughput.

This chapter is organised as follows: Section 4.2 describes the KNLMS algorithm (Richard, Bermudez, and Honeine, 2009), hyperparameter search, and literature review; Section 4.3 describes the proposed architecture; Section 4.4 shows the performance and accuracy results of the proposed architecture compared with CPU/GPU implementations; and conclusions are drawn in Section 4.5.

4.2 Background

4.2.1 Kernel Normalised Least Mean Squares

In this section, the KNLMS algorithm (Richard, Bermudez, and Honeine, 2009) is summarised with particular attention to aspects which affect hardware implementations. Also, some basic concepts of kernel methods are repeated from Section 2.1 in a way that highlights the key information which is relevant to this chapter.

In a standard supervised learning problem, training examples are input/output pairs $\{\mathbf{x}_i, y_i\}$, where $\mathbf{x}_i \in \mathbb{R}^M$ is the input vector and $y_i \in \mathbb{R}$ is the output or target. In regression, the goal is to estimate a function, $f(\mathbf{x}_i)$, which maps $\mathbf{x}_i \to y_i$. Kernel regression attempts to estimate this function by learning a dictionary $\mathcal{D}$, containing a subset of input vectors, and corresponding weights, $\boldsymbol{\alpha}$. A prediction, $\tilde{y}_i$, is then calculated as follows:

$$\tilde{y}_i = \sum_{n=1}^{N} \alpha_n \kappa(\mathbf{x}_i, \tilde{\mathbf{x}}_n), \tag{4.1}$$

where $\tilde{\mathbf{x}}_n$ is the n^{th} entry in $\mathcal{D}$, α_n is the n^{th} entry of $\boldsymbol{\alpha}$, N is the maximum size of $\mathcal{D}$, and $\kappa(\mathbf{x}_i, \mathbf{x}_j)$ is the kernel function, specified at design time. Although different kernels can be accommodated, in this work we focus on the commonly used radial basis function (RBF) kernel, $\kappa(\mathbf{x}_i, \mathbf{x}_j) = e^{-\gamma\|\mathbf{x}_i - \mathbf{x}_j\|^2}$, where γ is a free parameter chosen to suit the problem at hand.

The KNLMS algorithm is a stochastic gradient descent based algorithm which learns its model by taking small steps in the direction of the instantaneous gradient, to minimise the error in the current training example. Similar to algorithms such as the least mean squares (LMS) algorithm (Widrow and Hoff, 1960), it slowly converges to a solution over time.

The coherence criterion (Richard, Bermudez, and Honeine, 2009) is used to select the entries in the dictionary. For unit norm kernel functions, the coherence criterion is defined as follows: given a new input example at iteration t, $\mathbf{x}_t$ is added to the dictionary if $\max(|\mathbf{k}_t|) \leq \mu_0$, where $\mathbf{k}_t$ is the kernel vector with the n^{th} element being given by $\kappa(\mathbf{x}_t, \tilde{\mathbf{x}}_n)$, μ_0 is the coherence parameter chosen at design time. The weights for each iteration are then calculated by solving the instantaneous approximation to the following affine projection problem:

$$\min_{\boldsymbol{\alpha}} \|\boldsymbol{\alpha} - \hat{\boldsymbol{\alpha}}_{t-1}\|^2 \;\text{ subject to }\; y_t = \mathbf{k}_t^{\dagger}\boldsymbol{\alpha} \;\;, \tag{4.2}$$

where $\hat{\boldsymbol{\alpha}}_{t-1}$ is the set of weights obtained from the previous iteration and $\dagger$ denotes the vector transpose operation. Assuming that the current input vector, $\mathbf{x}_t$, can be adequately represented by the current dictionary and is not added to the dictionary, Equation (4.2) can be solved by minimising the following Lagrangian function:

$$J(\boldsymbol{\alpha}, \lambda) = \|\boldsymbol{\alpha} - \hat{\boldsymbol{\alpha}}_{t-1}\|^2 + \lambda(y_t - \mathbf{k}_t^{\dagger}\boldsymbol{\alpha}) \;\;. \tag{4.3}$$

A solution, $\hat{\boldsymbol{\alpha}}_t$, is found by differentiating Equation (4.3) with respect to $\boldsymbol{\alpha}$ and λ and setting the derivatives to zero, giving:

$$\begin{aligned} 2(\hat{\boldsymbol{\alpha}}_t - \hat{\boldsymbol{\alpha}}_{t-1}) &= \mathbf{k}_t\lambda \\ y_t &= \mathbf{k}_t^{\dagger}\hat{\boldsymbol{\alpha}}_t \;. \end{aligned} \tag{4.4}$$

Multiplying each term in the first equation by $\mathbf{k}_t^{\dagger}$ and substituting in for y_t, we get $\lambda = 2(\mathbf{k}_t^{\dagger}\mathbf{k}_t)^{-1}(y_t - \mathbf{k}_t^{\dagger}\hat{\boldsymbol{\alpha}}_{t-1})$. This yields the following recursive update equation:

$$\hat{\boldsymbol{\alpha}}_t = \hat{\boldsymbol{\alpha}}_{t-1} + \frac{\eta}{\epsilon + \mathbf{k}_t^{\dagger}\mathbf{k}_t}(y_t - \mathbf{k}_t^{\dagger}\hat{\boldsymbol{\alpha}}_{t-1})\mathbf{k}_t \tag{4.5}$$

where η is a step-size parameter and ϵ is a regularisation factor.

For the case where the current training example cannot be adequately represented by the dictionary, the current input, $\mathbf{x}_t$, is appended to the dictionary and the update equation becomes:

$$\hat{\boldsymbol{\alpha}}_t = \begin{bmatrix} \hat{\boldsymbol{\alpha}}_{t-1} \\ 0 \end{bmatrix} + \frac{\eta}{\epsilon + \mathbf{k}_t^{\dagger}\mathbf{k}_t}(y_t - \mathbf{k}_t^{\dagger}\begin{bmatrix} \hat{\boldsymbol{\alpha}}_{t-1} \\ 0 \end{bmatrix})\mathbf{k}_t \tag{4.6}$$

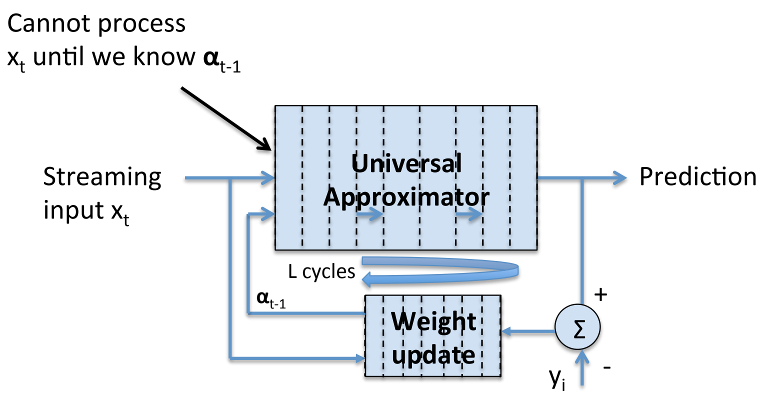


FIGURE 4.1: Pipelined implementation of a kernel adaptive filter.

Pseudocode for the KNLMS algorithm, adapted from (Richard, Bermudez, and Honeine, 2009) and (Yukawa, 2012), is shown in Algorithm 4.1.

Algorithm 4.1 KNLMS Algorithm with coherence criterion

```
Initialise the step-size, $\eta$, the regularisation factor, $\epsilon$,
the coherence parameter, $\mu_0$, and select a kernel function, $\kappa$.
Insert $\mathbf{x}_1$ into the dictionary, denote it as $\tilde{\mathbf{x}}_1$.
$\mathbf{k}_1 = \kappa(\mathbf{x}_1, \tilde{\mathbf{x}}_1)$, $\hat{\boldsymbol{\alpha}}_1 = 0$, n = 1.
while $t > 1$ do
    Get $\{\mathbf{x}_t, y_t\}$
    Calculate $\mathbf{k}_t = [\kappa(\mathbf{x}_t, \tilde{\mathbf{x}}_1), \cdots, \kappa(\mathbf{x}_t, \tilde{\mathbf{x}}_n)]^\dagger$.

    if $\max(|\mathbf{k}_t|) > \mu_0$ then
        Update $\hat{\boldsymbol{\alpha}}_t$ using Equation (4.5) $n = n + 1$.
        Append $\kappa(\mathbf{x}_t, \mathbf{x}_t)$ to $\mathbf{k}_t$.
        Insert $\mathbf{x}_t$ into the dictionary, denote it as $\tilde{\mathbf{x}}_n$.
        Update $\hat{\boldsymbol{\alpha}}_t$ using Equation (4.6).
    end if
end while
```

A simplified block diagram of a KAF implementation is shown in Figure 4.1. For KNLMS, the universal approximator block implements Equation (4.1), and the modify weights block, Equation (4.6). A high degree of pipelining is necessary for high throughput, but it should be evident that a new input cannot be processed until after the weights are modified. Assuming L cycles of latency from the input x_i to when the new weights have been determined, the next input cannot be processed until at least L cycles later, i.e. the initiation interval is L cycles.

4.2.2 Hyperparameter Search

In general, machine learning engineers make predictive models of data. When a machine learning engineer is confronted with a new dataset, they must choose the following: 1) which algorithm they will use to model the data; and 2) for a given algorithm, how to set its *hyperparameters*. In KNLMS, the dictionary, $\mathcal{D}$, and weights, α, are parameters which define the model, not hyperparameters which define the algorithm configuration. The parameters are learned from the training data, while the hyperparameters affect how a particular algorithm learns from that data.

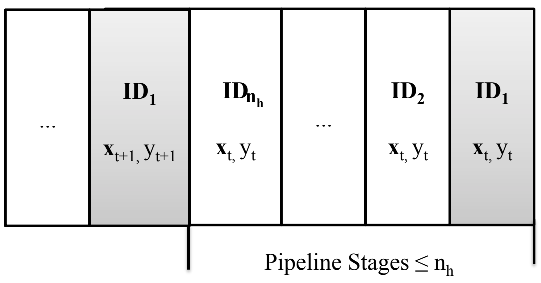


FIGURE 4.2: Required number of independent hyperparameter searches. In IDx, x represents a particular hyperparameter set.

The setting of hyperparameters has been shown to have significant impact on the modelling accuracy of a given algorithm. In particular, Cox and Pinto (2011) show that hyperparameter selection can be the difference between state-of-the-art or chance modelling accuracy. As such, to ensure that high modelling accuracy can be achieved, some sort of hyperparameter space exploration must be performed. In this work, we focus on a hyperparameter search, which seeks an optimised parameter set which minimises a cost function (Claesen and Moor, 2015), E. For regression problems, this cost function is often the least squares error function: $E = \sum_{n=0}^{N}(f(\mathbf{x}_n) - y_n)^2$. For classification problems, this is often the number of misclassified examples in a set. In Algorithm 4.1, the hyperparameter set is $\{\eta, \gamma, \mu_0, \epsilon\}$, these values being required by Equation (4.5), Equation (4.6) and the kernel function κ.

As previously mentioned, dependencies in Algorithm 4.1 exist on the update of $\hat{\alpha}_t$ and $\mathcal{D}_t$. However, if P parameters are tested at B different values, the number of hyperparameter sets to test is: $n_h = B^P$. A pipelined implementation with latency, L, less than or equal to n_h can thus be fully utilised by evaluating hyperparameter settings in different stages as shown in Figure 4.2. Put another way, hazards on contiguous inputs with the same ID in Figure 4.2 can be resolved by evaluating independent hyperparameter settings on successive cycles. For practical problems, this is almost always the case.

An alternative search algorithm uses random rather than evenly-spaced candidates (Pinto et al., 2009; Bergstra and Bengio, 2012a), e.g. for 4 hyperparameters, random search generates points randomly in a 4 dimensional space. This has the advantage that the experiment can be stopped at any time and the trials form a complete experiment, new trials can be added at any time, each trial is independent, and random search can be more efficient (Bergstra and Bengio, 2012a).

4.2.3 Literature Review

Kernel methods are eminently amenable to efficient hardware implementations and several implementations of SVM have been reported. Anguita et al. (2011) described an SVM core generator which allowed for different speed, resource and accuracy tradeoffs and utilised fixed point arithmetic. Papadonikolakis and Bouganis (2008) described a scalable SVM module generator which supported different kernel types. Their design supported different numbers of parallel computing tiles which allowed for performance/resource tradeoffs, and was partitioned into fixed point and floating point sections to achieve high performance while maintaining high accuracy. The MAPLE architecture as described by Majumdar et al. (2012) was designed to improve many learning algorithms, including SVM. MAPLE utilised two-dimensional vector

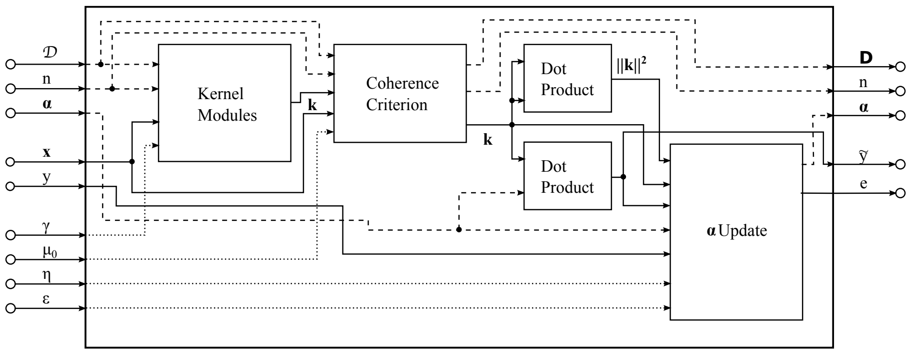


FIGURE 4.3: A block diagram of the KNLMS processor showing the various submodules.

processing elements to accelerate linear algebra routines. The architecture also supported off-chip memory, allowing it to accommodate large learning problems. While much of the computation is similar to KNLMS, as explained in the introduction, SVM is not suitable in online applications in which storage of the entire training set is undesirable, nor when incremental model updates are required.

Pang et al. (2013) proposed a compact and low latency microcoded soft vector processor. Parallelism was achieved using up to 128 floating-point vector processing elements, and kernel evaluations were accelerated through the inclusion of a hardware exponentiation unit. An implementation of the sliding window kernel recursive least squares (SW-KRLS) algorithm (Van Vaerenbergh, Via, and Santamaria, 2006) was used as an example in the work. The floating-point implementation of the quantised kernel least mean squares (QKLMS) algorithm (Chen et al., 2012) utilising the survival kernel (Chen, Zheng, and Principe, 2013) by Ren et al. (2014) is most comparable to this work. Differences include: (1) their work was limited to a single dimensional kernel vs arbitrary dimensions, (2) they employed the survival kernel compared to the much more commonly used Gaussian kernel in this design, (3) they achieve parallelism through pipelining and 128 parallel processing elements whereas our efficiency is by virtue of a fully pipelined design.

In this work, we avoid the dependency problem created by the recursive update expressions of KNLMS by filling the pipeline with independent models which are training during the hyperparameter phase of training a machine learning algorithm. Alternative approaches to avoiding this problem include: delayed model adaptation (Long, Ling, and Proakis, 1989), correction terms (Poltmann, 1995) and *braiding* (Tridgell et al., 2015).

As far as the author is aware, there are no specific FPGA based coprocessors for GPs. Although, general purpose machine learning accelerators (such as Majumdar et al. (2012)) and in particular, ones which include specific kernel function accelerators (such as Pang et al. (2013)) would most likely be able to accelerate GPs. Coprocessors may not have been implemented for GPs because of their close relationship to SVMs (Seeger, 2000) or perhaps simply because few hardware designers have knowledge of GPs.

Delayed model adaptation has been applied effectively to the least mean squares (LMS) algorithm (Widrow and Hoff, 1960). The resultant delayed LMS (DLMS) algorithm (Long, Ling, and Proakis, 1989) has been shown to be stable under certain configurations and that high frequency FPGA implementations can be achieved (Yi et al., 2005). Since introducing a model adaptation delay changes the learning characteristics of an algorithm, it may not be suitable for hyperparameter optimisation for the following reasons: (1) as far as the author is aware, no stability analysis has been done for a version of LMS which uses a kernel function; (2) the hyperparameters found using a delayed version of KNLMS may differ significantly from KNLMS itself, i.e., the learned hyperparameters would need to be deployed on a system which also implements KNLMS with a model adaptation delay; and (3) the computational complexity of KNLMS is higher than LMS, as such the number of pipeline stages (see Section 4.4) required to achieve high clock frequencies would most likely cause undesirable learning behaviour.

Correction terms are a way of rearranging the equations of a recursive algorithm in order to reduce the critical path of the dependency loop. They have been applied to the LMS algorithm (Poltmann, 1995) and utilised effectively in hardware to create high frequency implementations of LMS (Douglas, Zhu, and Smith, 1998) which maintain the same learning characteristics of LMS. Adding correction terms usually introduces redundant calculations (when compared with a serial implementation.) Also, the amount of extra computations required is proportional to the number of pipeline stages used and as such, would significantly reduce the maximum dictionary size / feature length that can be realised by our core.

Braiding (Tridgell et al., 2015) was used effectively to overcome the dependencies present in the Naive Online regularised Risk Minimisation Algorithm (NORMA) (Kivinen, Smola, and Williamson, 2004), and endeavours to optimise latency rather than throughput which is the topic of the present work. It can be seen as an extension of correction terms to kernel based online learning algorithms when the dictionaries of such algorithms are created from a sliding window. The KNLMS algorithm used in this work does not have this property for the entries of its dictionary, rather it uses the coherence criterion (Richard, Bermudez, and Honeine, 2009) to select its entries. However, it is possible the technique could be extended to KNLMS, but braiding also suffers from an increase in calculation requirements proportional to the number of pipeline stages in the circuit. Similar to correction terms, this would likely significantly limit the problem sizes that could be tackled by our core.

4.3 Architecture

In this section, our fully pipelined KNLMS architecture is described, highlighting areas suitable for optimisation. In addition, scalability of the design is explored.

Referring to Algorithm 4.4, our main insight is to reorder the commonly used search loop on the left to that on the right. This allows the independent hyperparameter evaluations to fill the pipeline and avoid dependencies, as previously explained in Section 4.2.2. Note that n_h is the numbers of hyperparameter sets to be evaluated.

4.3.1 High Level Description

The idea behind the design is to create a module to accelerate the operations required to update the kernel regression model from time step $t-1$ to t, i.e. those within the while loop of Algorithm 4.1. We call this the *forward path*. In doing this, we completely unrolled the algorithm and instantiated dedicated hardware for every operation within the algorithm. For scaling the design, we kept the same basic structure of this unrolled, dataflow implementation of the design, but folded any dot products to be executed over multiple cycles. A block diagram showing the basic structure of the processor is shown in Figure 4.3. The submodules are responsible for the following functions: (1) the kernel modules calculate the kernel vector, $\mathbf{k}_t$; (2) the coherence

Algorithm 4.4 Difference between a regular parallel search and a pipelined search.

Initialise hyperparameter sets Λ_{1-n_h}, training data $TR_{1-n_{TR}}$, and testing data $TST_{1-n_{TST}}$

$\boldsymbol{\alpha}_{\Lambda_i}$: Weights trained with Λ_i,

$\mathbf{k}_{j(\Lambda_i)}$: a kernel vector produced from TST_j and dictionary trained with Λ_i

```
Normal Grid/Random Search:
begin
    Initialise the sums of errors $E_{1-n_h} = 0$
    for $i = 1 : n_h$ do
        for $j = 1 : n_{TR}$ do
            Execute Algorithm 4.1 using $\Lambda_i$ with $TR_j$
        end for
        for $j = 1 : n_{TST}$ do
            $E_i += (y_j - \mathbf{k}^{\dagger}_{j(\Lambda_i)} \boldsymbol{\alpha}_{\Lambda_i})^2$
        end for
    end for
    Pick $\Lambda_\beta$, where $\beta = argmin_\beta E_\beta$
end
```

```
Pipelined Search:
begin
    Initialise the sums of errors $E_{1-n_h} = 0$
    for $i = 1 : n_{TR}$ do
        for $j = 1 : n_h$ do
            Execute Algorithm 4.1 using $\Lambda_j$ with $TR_i$
        end for
    end for
    for $i = 1 : n_{TST}$ do
        for $j = 1 : n_h$ do
            $E_j += (y_i - \mathbf{k}^{\dagger}_{i(\Lambda_j)} \boldsymbol{\alpha}_{\Lambda_j})^2$
        end for
    end for
    Pick $\Lambda_\beta$, where $\beta = argmin_\beta E_\beta$
end
```

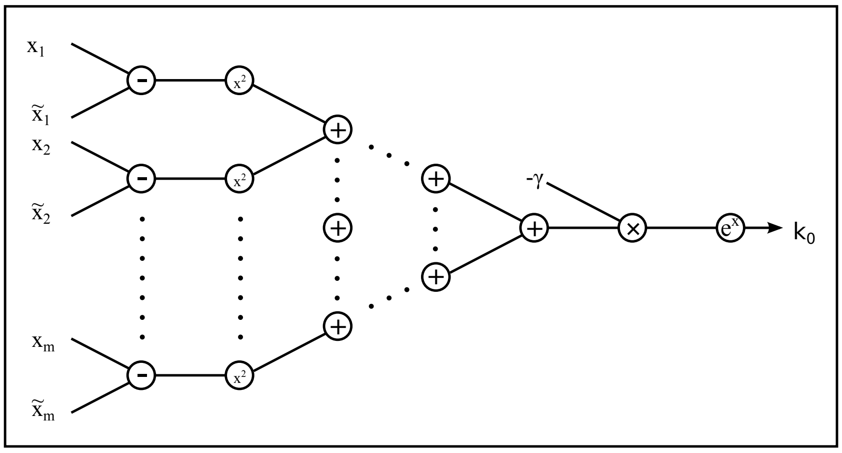


FIGURE 4.4: Dataflow diagram of a kernel module.

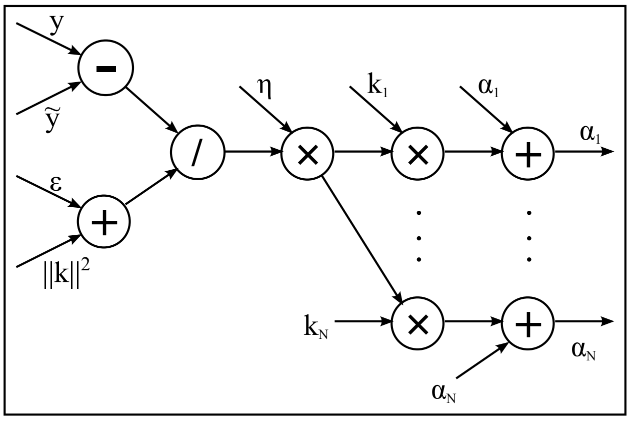


FIGURE 4.5: Dataflow diagram of the $\boldsymbol{\alpha}$ update module.

criterion module decides whether to add the latest input example to the dictionary, and updates it if necessary; (3) the dot product modules, which produce the a-priori estimate of y_t, denoted by $\tilde{y}_t$, and the normalisation term, $||\mathbf{k}_t||^2$; and (4) the $\boldsymbol{\alpha}$ update module produces the updated weights, $\boldsymbol{\alpha}_t$. In order to compute multiple iterations of the KNLMS algorithm, the forward path module is controlled by a scheduler, which is described in Section 4.3.5.

4.3.2 Kernel Module

Figure 4.4 shows the dataflow graph of a kernel module. The kernel module computes the Gaussian kernel, given by: $\kappa(\mathbf{x}_i, \mathbf{x}_j) = e^{-\gamma||\mathbf{x}_i - \mathbf{x}_j||^2}$. Each kernel module requires $2M - 1$ adders, $M + 1$ multipliers and 1 exponential unit, where M is the feature length. N kernel modules are required for a design supporting a maximum dictionary size of N. The most computationally expensive part of the KNLMS processor is the calculation of the kernel vector.

4.3.3 Alpha Update Module

The $\boldsymbol{\alpha}$ update module finishes the training step by calculating $\boldsymbol{\alpha}_t$ as shown in Equation (4.5). The dataflow graph for the $\boldsymbol{\alpha}$ update module is shown in Figure 4.5. The $\boldsymbol{\alpha}$ update module first calculates the prediction error and the normalisation term. This is followed by a scalar vector product and an elementwise vector addition. The $\boldsymbol{\alpha}$ update module operates on vectors of length N and as such, requires $N + 1$ multipliers, $N + 2$ adders and 1 divider.

4.3.4 Coherence and Dot Product Modules

The coherence module is a simple control module. It takes $\mathbf{k}_t$, $\mathbf{x}_t$, μ_0, n and $\mathcal{D}$ as inputs. $\mathbf{k}_t$ is padded with zeros for each unused entry in $\mathcal{D}$. If $\max(|\mathbf{k}_t|) \leq \mu_0$, then n becomes $n + 1$ and $\mathbf{x}_t$ is appended to $\mathcal{D}$. Otherwise, n and $\mathcal{D}$ are unchanged.

The two dot product modules are made using parallel multipliers followed by an adder tree. Each module operates on vectors of length N and as such, require N multipliers and $N - 1$ adders.

4.3.5 Optimisations

In order to maximise performance, the optimisations described in this subsection were implemented.

A fully pipelined design cannot be directly synthesised from a C++ description of the algorithm in Algorithm 4.1, due to the dependency of the updated dictionary $\mathcal{D}$ and weights $\hat{\boldsymbol{\alpha}}_t$ on the new kernel vector $\mathbf{k}_t$. By omitting the update steps, we can turn the datapath into an acyclic one. The feedback connection is then made by externally connecting outputs to corresponding inputs in Figure 4.3. This results in the desired design with initiation interval of 1. Dictionary and weight updates are delayed by L, the latency of the KNLMS core.

A significant bottleneck in creating accurate machine learning models is parameter optimisation. Even if the kernel function is fixed to be the Gaussian kernel, a search is required over the hyperparameters. This involves performing regression over a test data set using different parameter settings. Since these are independent problems, they can be executed in parallel as n_h independent tasks. Each task is executed in a different pipeline slot so all hardware units in the KNLMS forward path evaluation pipeline can be fully utilised.

Hyperparameter search using KNLMS requires control logic and BRAMs beyond that of Figure 4.3. In order to perform regression on n_h independent problems, we require storage of $n_h\times$ dictionaries of size MN, and $n_h\times$ length-M weight vectors. This is achieved by using block RAMs and indexing them with a counter $l \in [0, \ldots, n_h)$ so that every n_h cycles, we return to the same dictionary and weight vector. This arrangement removes the need for an $n_h : 1$ multiplexer per dictionary and weight entry. Since writing and reading dictionary and weight vector entries is required each cycle, dual-port BRAMs are used. The architecture produces an error e, and updates $\mathcal{D}$, α, and n every cycle.

One further improvement over (Fraser et al., 2015a) is support for random search. A parameter memory (as shown in in Figure 4.7) per hyperparameter allows runtime downloading of parameter choices to be searched. If initialised to grid points, a grid search is realised; whereas if the parameter memory is initialised to random points, a random search will be performed. An alternative approach might be to replace the parameter memory with a state machine that generates grid or random points. In practice, the current architecture is usually preferable since it is more flexible and the number of search points per parameter is modest.

Computing the kernel function $\kappa(\mathbf{x}_i, \mathbf{x}_j)$ for the Gaussian kernel requires an M-input adder tree which has a total latency of $\lceil \log_2 M \rceil$ times the latency of a single adder. We observe that: (1) the inputs to this adder are strictly positive, so unsigned arithmetic can be used; (2) the output is passed through a function $e^{-\gamma \sum x^2}$ which is not sensitive to small changes in the input; and (3) computation can be done in fixed point. This can reduce latency and allow accuracy-speed tradeoffs.

4.3.6 Implementation of the KNLMS Core

For this work, Vivado High Level Synthesis (HLS) was used to implement the core. Vivado HLS was used because it allows the core designer to explore a large space of possible designs while making only small changes to the code. Also, if there are no data dependencies present within the C++ used to describe the core, Vivado HLS will automatically pipeline the core in an attempt to meet the desired clock constraint.

Firstly, the forward path of the KNLMS algorithm was implemented as a C++ function which takes the hyperparameters, the model parameters ($\mathcal{D}$, $\boldsymbol{\alpha}$), and a single training example as inputs. As outputs, the function returns the updated weights, dictionary and an a priori prediction for the training example. Unlike some other implementations of adaptive filters (such as Douglas, Zhu, and Smith (1998) and Yi et al. (2005)) the model parameters are not *stored* within the core itself, rather they are passed as inputs to the core. As such, there are no dependencies within the core which means it may be pipelined arbitrarily without affecting the accuracy of the resultant updated model.

The top level module (in HLS), sits outside the core. It uses BRAMs to store the different parameter configurations and the corresponding models, i.e., the dictionaries $\mathcal{D}$, and weights, $\boldsymbol{\alpha}$. The module then feeds each training example, parameter configuration and model into the core while ensuring that the same parameter configuration does not exist within the pipeline of the KNLMS core multiple times.

4.3.7 System Implementation of Hyperparameter Search

The floating-point KNLMS core can be integrated with a PCIe interface as illustrated in Figure 4.6. Data ingress is controlled by a FIFO, and a n_h-word parameter memory ($n_h \geq L$) for each of the 4 parameters shown in the bottom left module of Figure 4.3 was used to store the parameters to be searched. Separate memories indexed by l (as detailed in Section 4.3.5) were used to store dictionary and weight values.

When the input FIFO becomes non-empty, the serial to parallel converter converts the data to a vector. A sequence of n_h independent optimisations with different parameter values is then streamed through the KNLMS processor. In our previous implementation (Fraser et al., 2015a), an output FIFO was used to collect n_h output values per input vector. In such an implementation, care must be taken to limit the number of inputs sent before the outputs are read. While this avoids deadlocks, it also limits the burst transfers made on the PCI bus and reduces performance.

An improved system implementation, illustrated in Figure 4.7, avoids transfers from the KNLMS processor by accumulating the squared error values on-chip. The host code simply writes the parameters and training set to the core via the RIFFA2 (Jacobsen et al., 2015) interface, and reads back an expected number of words from the same interface. This allows all of the training input data to be transferred to the core with a single data stream transfer RIFFA API function call for each training-validation set. Therefore, this improved system implementation minimises the overhead from the blocking operations between sending and receiving data for the RIFFA interface. We implemented a 2-fold cross-validation platform with the architecture of Figure 4.7. A subsequent read of n_h sum of squared error values is required to determine which set achieved the lowest error. The n_h numeric values from the system implementation were verified, comparing to the results from the C implementation. In summary, these changes allowed for more efficient use of the available PCI bandwidth through:

1. reducing the amount of data transferred between the host and the accelerator; and

2. changes to the host code, allowing larger burst sizes to be used during memory transfers.

4.3.8 Arithmetic

Two implementations were made, one using floating-point and the other fixed-point. The former has advantages of large dynamic range and ease of comparison with

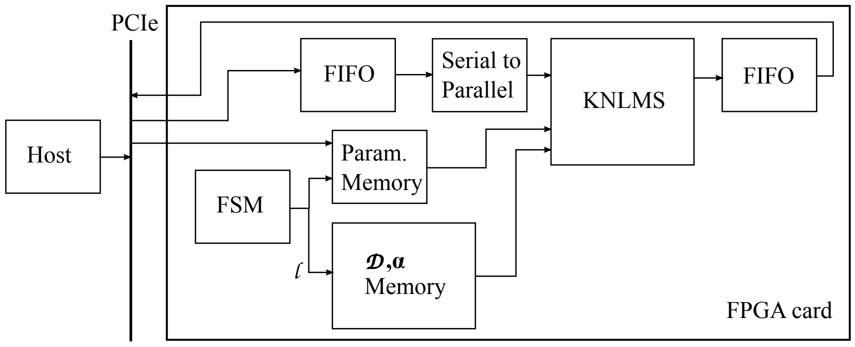


FIGURE 4.6: Block diagram illustrating system integration of the KNLMS processor.

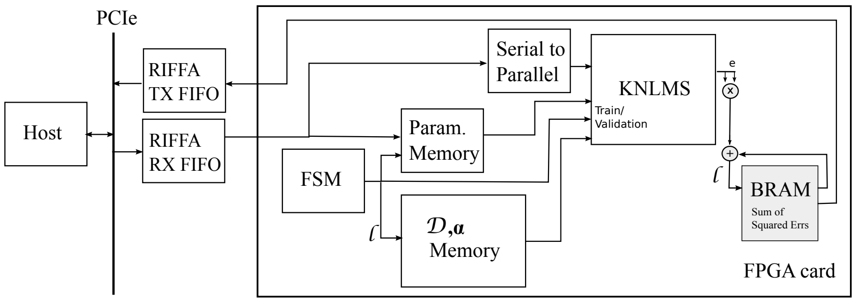


FIGURE 4.7: Block diagram illustrating improved system implementation of the KNLMS processor.

microprocessor-based implementations, whereas the fixed-point implementation offers lower latency and smaller area.

Implementation of the basic operators in fixed point arithmetic is straightforward. In this work, we chose a 21-bit two's complement fixed-point representation (5 integer bits and 16 fraction bits) with truncated rounding and saturating arithmetic. As shown later, this is sufficient precision to obtain comparable results with single-precision floating-point.

Hardware architectures for evaluating the exponential function have been previously reported (Jamro, Wiatr, and Wielgosz, 2007; Wielgosz, Jamro, and Wiatr, 2008; Detrey and Dinechin, 2005; Pottathuparambil and Sass, 2009; Alachiotis and Stamatakis, 2011). We employed a look-up table based approach presented in (Wielgosz, Jamro, and Wiatr, 2008), without the Taylor series expansion. The approximation is based on the mathematical property

$$e^x = 2^{x \cdot log_2 e} = 2^{x_I} \cdot 2^{x_F}, \tag{4.7}$$

where x_I is the integer part and x_F is the fraction part of $x \cdot log_2 e$. Since we only use e^x in computing the Gaussian function, $e^{-\gamma \|\cdot\|^2}$, $x = -\gamma\| \cdot \|^2 \leq 0$. Hence, the input domain for Equation (4.7) can be expressed using non-negative numbers x^+, i.e., x = $-x^+$. We further represent x^+ in terms of its integer and fractional parts so that $x^+ = x_I^+ . x_F^+$, and break x_F^+ into most significant ($x^+_{F_{MSB}}$) and least significant ($x^+_{F_{LSB}}$) parts to give

$$e^x = 2^{x_I} \cdot 2^{x_F} = 2^{-x_I^+} \cdot 2^{-x^+_{F_{MSB}}} \cdot 2^{-x^+_{F_{LSB}}}. \tag{4.8}$$

Figure 4.8 illustrates our implementation using two lookup tables. The width for both $F_{MSB}[x_+ \cdot log_2 e]$ and $F_{LSB}[x_+ \cdot log_2 e]$ was chosen to be 8 bits. Consequently, the

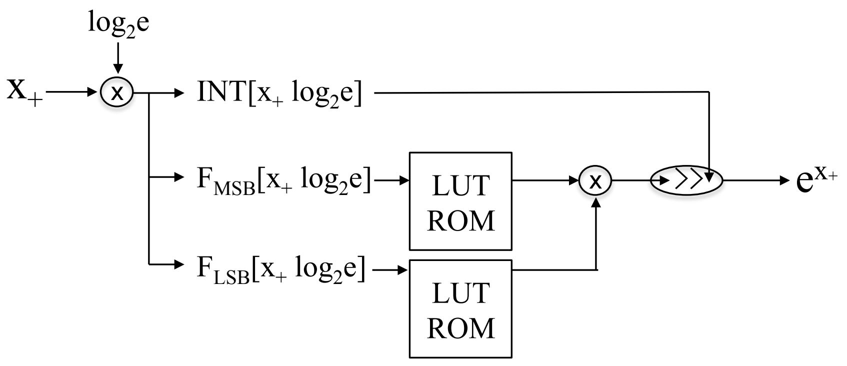


FIGURE 4.8: Dataflow of the exponential function module

TABLE 4.1: Formulae for the number of floating-point operators required and latency in cycles for (Float, Fixed: 5 bits for integer and 16 bits for fraction with saturating arithmetic).

	+ (11,3)	× (7,6)	/ (30,41)	exp (20,9)	< (4,1)
Operation	$2MN + 2N$	$MN+$ $4N + 1$	1	N	$N - 1$
Latency	$\log_2 N+$ $\log_2 M + 3$	5	1	1	$\log_2 N$

size of each of the two LUTs is 2^8. N exponentiation modules are required in a fully pipelined KNLMS design.

The fixed-point implementation produces an error less than or equal to one half of one unit in the last place (Detrey and Dinechin, 2005; Wilkinson, 1994), i.e.

$$|e^x - \hat{e}^x| = 2^{-x_I^+} \cdot |2^{-x_F^+} - \hat{2}^{-x_F^+}| \leq \xi \tag{4.9}$$

where l is the number of bits in x_F, $\xi = 2^{-(l+1)}$ is a machine epsilon in a fixed point data format and $\hat{e}^x$ is the exponential estimation of the implementation.

4.3.9 Area and Latency

We estimated the scalability of the architecture with the key parameters N and M. The number of required operators and estimated latency is shown in Table 4.1. The operator latency is given in parentheses next to the operator symbol. In order to estimate the latency for a given design, the operator latency is multiplied by the expression in the latency row. In terms of worst case scalability, the area of arithmetic operators is $\mathcal{O}(MN)$, memory usage is $\mathcal{O}(MN)$, and latency $\mathcal{O}(\log_2 N + \log_2 M)$.

In the following section, linear regression is used to model area (LUTs and DSPs) and latency (L) using the formulae

$$\begin{aligned} LUTs &= l_1 MN + l_2 N + l_3 & (4.10)\\ DSPs &= d_1 MN + d_2 N + d_3 & (4.11)\\ L &= L_1 \log_2 MN + L_2 \log_2 N + L_3 & (4.12)\end{aligned}$$

TABLE 4.2: Summary of place and route output for each of the KNLMS designs

	Naïve (N=16)	Float (N=16)	Fixed (N=16)	Fixed (N=32)
BRAM18K (2060)	8 (0.4%)	145 (7.0%)	138 (6.8%)	274 (31.3%)
DSP48 (2800)	12 (0.4%)	1267 (45.3%)	210 (7.5%)	418 (14.9%)
LUTs (304K)	4,550 (1.5%)	150,494 (49.5%)	53,273 (17.3%)	109,546 (36.2 %)
Latency (cycles)	756	207	121	124
II (cycles)	757	1	1	1
$F_{max}(MHz)$	96.7	314	286	206
GOPS	0.07	161.1	146.7	211.2

4.4 Results

This section describes the resource utilisation, performance and accuracy of the implementation written in C. The design was synthesised and implemented using Xilinx Vivado HLS 2015.3. The target platform was a Xilinx VC707 evaluation board using a Xilinx Virtex 7 XC7VX485TFFG1761-2 FPGA.

4.4.1 Synthesis and PAR Results

Table 4.2 shows place and route (PAR) results for the four different implementations of Figure 4.3: (1) Naïve - an unoptimised Vivado HLS synthesised implementation derived from KAFBOX (Van Vaerenbergh, 2012), representing a design without consideration of the resulting hardware datapath; (2) Float - a single precision floating point design which uses the Xilinx floating point cores throughout and contains all optimisations described in Section 4.3.5; and (3) Fixed (5 integer bits and 16 fraction bits) - with all optimisations described in Section 4.3.8.

Floating-point operations are single precision and IEEE-754 compliant with the exception that denormalised numbers are not supported. Although our design is parameterised, results for the settings $N = 16$ and $M = 8$, are reported unless stated otherwise. The GOPS are estimated using F_{max}, the initiation interval (II) and the number of operations required for single update. With reference to Table 4.1, the number of operations is 513. Note that for the Naïve and Float designs, GOPS is equivalent to GFLOPS.

Table 4.3 shows the relationship between the design parameters M and N, and the latency and hardware resources. These numbers are different to Table 4.2 because they are post-synthesis estimates rather than place and route results.

Using the formulae in Sections 4.3.9 to 4.3.9, linear regression was applied to obtain linear models. The estimates are given in parentheses in Table 4.3 and a maximum percentage error of less than 10% was observed. This confirms that the simple area and latency models in Table 4.1 are capable of accurate prediction.

As expected, Fixed requires fewer resources, and can support a larger dictionary. For $N = 32$ and $M = 8$, Fixed achieved 211 GOPS (1,025 operations at 206 MHz) and only used 36% of LUTs, 15% of DSPs and 31% of BRAMs.

4.4.2 Learning Accuracy with a Grid/Random Search

We examine out-sample error for our implementations for the chaotic MG-30 Mackey-Glass benchmark modelling the differential equation $dx(t)/dt = ax(t-\tau)/(1+x(t-\tau)^{10}) - bx(t)$ with $(a = 0.2, b = 0.1, \tau = 30)$, as implemented in KAFBOX (Van

TABLE 4.3: Area utilisation of different designs obtained from synthesis. Estimates obtained via linear regression are in parentheses.

Type	M	N	LUTs (Est)	DSPs (Est)	L (Est)	F_{max}
Float	2	16	77K (77K)	595 (595)	185 (185)	385
	4	16	109K (109K)	819 (819)	196 (196)	385
	16	16	307K (307K)	2163 (2163)	218 (218)	385
	8	2	23K (25K)	161 (161)	162 (162)	385
	8	4	46K (47K)	319 (319)	177 (177)	385
	8	8	95K (90K)	635 (635)	192 (192)	385
	8	16	173K (175K)	1267 (1267)	207 (207)	385
Fixed	2	16	31K (31K)	196 (197)	112 (109)	321
	4	16	47K (47K)	260 (261)	115 (116)	321
	16	16	142K (142K)	644 (645)	133 (130)	321
	8	2	13K (14K)	54 (59)	108 (107)	321
	8	4	23K (24K)	106 (106)	111 (112)	321
	8	8	47K (42K)	210 (201)	116 (118)	321
	8	16	78K (79K)	388 (389)	121 (123)	321

Vaerenbergh, 2012). A training set of 3,999 samples was used for hyperparameter search with 3-fold cross validation. We use 4 hyperparameter candidates for each hyperparameter in a grid search (i.e, $n_h = 4^4$) and 256 random hyperparameter sets in a random search. The ranges for parameter sets for grid and random parameter searches are set as follows: $\gamma = [0.01, 2]$, $\eta = [0.05, 0.3]$, $\epsilon = [0.001, 0.5]$ and $\mu_0 = [0.5, 0.8]$. The result of this execution was two hyperparameter sets.

The quality of the hyperparameters were tested on 100, 1100 out-sample data points, which we call the testing sets. On each of the 100 testing sets, we perform the following experiment: 1) split the 1100 samples up into 1000 training points and 100 validation points; 2) train a KNLMS model on a single training point; 3) after training on a single point, test the models prediction accuracy on the 100 validation points and calculate the mean squared error (MSE); and 4) repeat steps 2) and 3) until all training points are trained. After calculating the above steps on each of the 100 testing sets, for float and fixed point, with the fractional length, $FL = 8$ and $FL = 16$, we then calculate the average MSE at each training iteration, as well a average relative error and maximum relative error with respect to floating point. The left plot of Figure 4.9 shows the average MSE of our KNLMS configurations using the hyperparameters found using the abovementioned methodology. Explicitly, the parameters $(\sigma, \eta, \epsilon, \mu_0)$ found were $0.85, 0.3, 0.001, 0.8$ for the grid search, and $0.5857, 0.1912, 0.3907, 0.7815$ for the random search. The values on the x-axis represent the current training iteration and the y-axis represents the average MSE over the validation set. On the right, the relative MSE is shown between float and fixed point with $FL = 8$ and $FL = 16$. Bit-accurate Vivado HLS C simulations were used to obtain this figure. When $FL = 16$ it is clear that float and fixed point produce almost identical results with less than 1/1,000 relative error for all data points and on average 1/10,000 relative error. While when $FL = 8$, the average relative error becomes over 1/100 and the maximum relative error over 1/10. In our experiements, random search resulted in a lower error than grid search, indicating a better hyperparameter set was found. This observation is consistent with others studies which found random search to often superior to grid search (Bergstra and Bengio, 2012a). Based on Figure 4.9, 16 bits are enough for

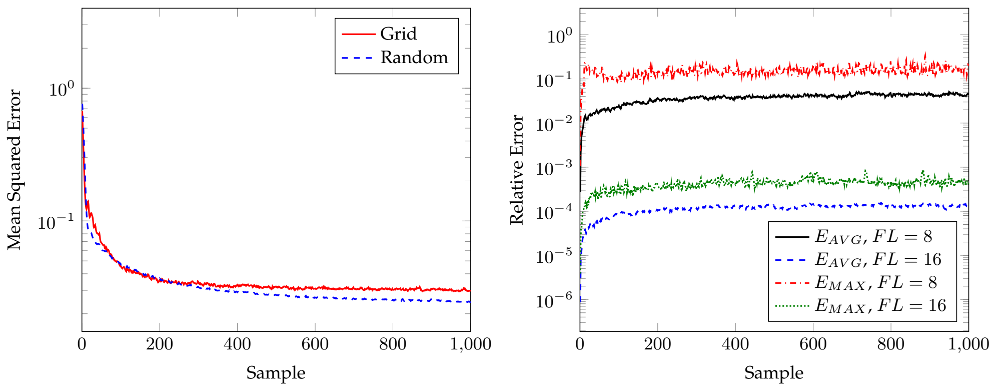


FIGURE 4.9: Accuracy: grid search vs random search using float (left.) Average relative and maximum relative error fixed point with different fractional widths compared to float (right.)

Fixed fraction width to produce almost identical learning accuracies to Float. In this work, we employed a dictionary size of $N = 16$ for our Float design due to resource limitations. Since Fixed uses less area, the same FPGA can support a larger dictionary size of $N = 32$. In some cases, it may be possible that fixed can achieve improved accuracy by employing a larger dictionary.

We end this subsection by noting that these results are data dependent and it is not possible to make any generalisations for other data sets. Moreover, insight into how precision requirements, hyperparameter values and dictionary size affect the quality of predictions is a research problem beyond the scope of this work and, in the opinion of the authors, not currently tractable for non-linear KAFs.

4.4.3 KNLMS Processor Performance

A comparison of performance with other KAF implementations is challenging since previous work implemented different algorithms. SW-KRLS (Van Vaerenbergh, Via, and Santamaria, 2006) requires a matrix inversion per update and has $\mathcal{O}(N^2 + NM)$ time complexity. This is in contrast to KNLMS which is $\mathcal{O}(NM)$ and uses stochastic gradient descent techniques (Liu, Príncipe, and Haykin, 2011). Moreover, one should be careful in comparing Altera and Xilinx LUTs and DSP blocks as they are different. Nevertheless, a summary of previous online KAF implementations of which we are aware is presented in Table 4.4. Clearly, the different versions of our KNLMS processor have much higher throughput when compared to the other implementations.

The CPU (C) and GPU (CUDA) are versions of KNLMS with a parallelised parameter search implemented using C/CUDA for CPU/GPU respectively.[1] For the CPU (C) version, many linear algebra libraries were tested including: BLAS (Lawson et al., 1979), ATLAS (Whaley and Petitet, 2005) (with and without multithreading), OPENBLAS (Xianyi, Qian, and Chothia, 2014) and hand coded routines. The fastest version was chosen to populate Table 4.4, which to our surprise, was our hand coded routines. We suspect that the vector sizes (define by $M \leq 8$, $N \leq 16$) were not large enough to take advantage of cache blocking, and the vectorisation methods provided by ATLAS/OPENBLAS, while aggressive compiler optimisations caused the hand coded routines to be inlined, removing the function call overhead associated with an external library. A parallel parameter search was implemented using OpenMP (Sato, 2002) on the outer loop described in Algorithm 4.4 (the normal grid search). Similar to

[1]Software available at: `https://bitbucket.org/nick_fraser/libkaf/`

TABLE 4.4: Comparison of online kernel method implementations

Implementation	Algorithm	Device	M	N	DSPs	LUTs	BRAM	Freq MHz	Time ns	Slowdown rel. to Float
Naïve	KNLMS	Xilinx XC7VX485-2	8	16	12	4550	8	96.7	7,829	2,462
Float	KNLMS	Xilinx XC7VX485-2	8	16	1267	150,494	145	314	3.18	1
Fixed	KNLMS	Xilinx XC7VX485-2	8	16	210	53,273	138	286	3.50	1.1
System (Float)	KNLMS	VC707 dev board	8	16	1272	142,900	229	250	4	1.3
System (Fixed)	KNLMS	VC707 dev board	8	16	226	54,689	230	125	8	2.5
CPU (C)	KNLMS	Intel Xeon E5-2670	8	16	-	-	-	2,600	41	12.89
GPU (CUDA)	KNLMS	Nvidia GRID K520	8	16	-	-	-	800	11	3.46
Pang et al. (2013)	SW-KRLS	Altera Stratix V 5SGXEA7C2	7	16	30	41,476	227	237	9,000	2,830

the CPU (C) code, the GPU (CUDA) implements the KNLMS parameter search loop described in Algorithm 4.4 (the normal grid search). Each GPU thread implements KNLMS training on a different set of parameters in the parameter search. For both CPU and GPU parameter searches, great care was taken to ensure that as many variables as possible were private to each thread. Also, for both CPU and GPU parameter searches, the number of parameters and the length of the training set were scaled to find the best average time per prediction. As stated in Section 4.2, Pang et al. (2013) implement a different kernel adaptive filtering algorithm using a microcoded vector processor, rather than a fully parallel, fully pipelined implementation described herein. The goal of the work from Pang et al. (2013) is to optimise latency, rather than throughput and as such, their performance is much lower than ours. However, their work constitutes the highest performing FPGA implementation of a KAF available in the literature and therefore, is included in Table 4.4.

4.4.4 System Performance

RIFFA 2.2.0 (Jacobsen, Freund, and Kastner, 2012) was used to provide high speed data communication between our host computer and the FPGA via a PCIe GEN2 bus. The VC 707 board used supports a maximum bandwidth 4 GB/s for uni-directional transfers and RIFFA could reach around 80% of this value. However, high bandwidth can only be achieved if sufficient data is sent to amortise transaction overheads. The optimised interface requires modest I/O bandwidth because each input vector is used $n_h = 256$ times before the next one is required.

Using the optimised system interface, with both the KNLMS Float core and PCI interface operating at 250 MHz, the same MG-30 benchmark set samples were trained over a parameter space of $n_h = 256$ values. Figure 4.10 shows the performance as a percentage of the peak performance where $f = 250\ MHz$ is the operating frequency, while varying the training set size.

While the design in (Fraser et al., 2015a) only achieved 23% of the highest achievable performance for the core, our new design can achieve full performance for training sets of more than 1000 input vectors. The main sources of inefficiencies in (Fraser et al., 2015a) can be attributed to turnaround time of the PCIe bus (i.e. transfers were made in small blocks to avoid filling the FIFO and the result was read into the host before the next block was sent), these were overcome in the present design by calculating the sums of errors in the FPGA, rather than the host. This significantly reduced the amount of data that needed to be sent from the FPGA to the host. The entire data set could be sent in a single transfer without filling the FIFOs, as detailed in Section 4.3.6.

Note that, the system implementation used a different clock constraint in Vivado HLS in order to meet the timing requirements of the 250MHz clock on the VC707 dev

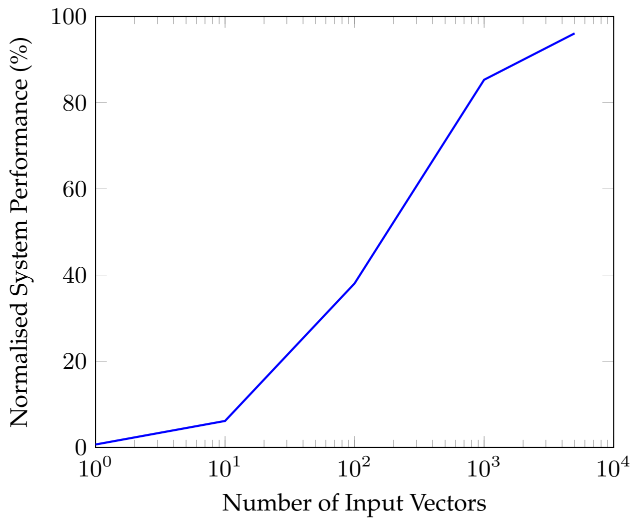


FIGURE 4.10: Plot illustrating system performance of the KNLMS processor vs training set size.

TABLE 4.5: Scaling the KNLMS core to larger FPGAs and larger problem sizes

Implementation	Device	M	N	DSPs	LUTs	Freq MHz	II	Latency	Time ns
Float	Xilinx XC7VX485-2	8	16	126	39,351	283	16	331	3.53
Float	Xilinx XC7VX485-2	8	32	247	77,195	283	16	475	3.53
Float	Xilinx XC7VX485-2	8	64	441	175,031	283	16	772	3.53
Float	Xilinx XC7VX485-2	8	128	882	394,130	283	16	1348	3.53
Float	Xilinx XC7VX485-2	8	256	1769	977,716	283	16	2500	3.53
Float	Xilinx XC7VX485-2	8	32	222	70,924	246	32	483	4.06
Float	Xilinx XC7VX485-2	8	64	439	138,254	246	32	771	4.06
Float	Xilinx XC7VX485-2	8	128	405	291,576	246	32	1367	4.06
Float	Xilinx XC7VX485-2	8	256	808	682,692	246	32	2519	4.06
Float	Xilinx XC7VX485-2	8	64	96	112,487	246	64	994	4.06
Float	Xilinx XC7VX485-2	8	128	216	224,675	240	64	1386	4.17
Float	Xilinx XC7VX485-2	8	256	432	540,213	240	64	2538	4.17
Device	Xilinx XC7VX485-2	-	-	2800	303,600	-	-	-	-

board. When synthesised without the PCI interface, the Fixed and Float were capable of clock frequencies of 314 MHz and 286 MHz respectively, however, inclusion of the RIFFA core caused the maximum frequencies to drop to 250 MHz and 125 MHz. Separate clocks for the core and interface should support a higher KNLMS core clock rate and a further improvement in throughput.

4.4.5 Scaling the KNLMS Core

In all previous sections, we only considered a fully parallel, fully spatialised implementation of the KNLMS core. In this subsection, we consider how the core will scale to larger FPGAs and also larger problem sizes. In order to handle larger problem sizes, we fold the design over the maximum dictionary size, N. We use initiation intervals (IIs) of 16, 32 and 64 while scaling the maximum dictionary size. The performance and area estimates, shown in Table 4.5 were created using presynthesis results from Vivado HLS. For the most part, increasing the II by a factor of 2 allows a dictionary which is twice as large to be accommodated for a similar area cost. Surprisingly, for high initiation intervals and dictionary sizes, the DSP usage is lower for the same N/II ratio. In the case of the Xilinx XC7VX485-2 device, the largest problem that can fit, based on Table 4.5, is $N = 128$ at a folding factor of 64. In practice, values of M and N can vary significantly to meet the demands of the specific application and the selected hyperparameters. Richard, Bermudez, and Honeine (2009) report sizes for $M \leq 10$ and as low as $M = 2$, while values for N range from $5 \leq N \leq 20$. For other applications of kernel methods, sizes for M, N can range into the tens, hundreds and even thousands (Engel, Mannor, and Meir, 2004).

4.5 Conclusion

A fully pipelined FPGA implementation of hyperparameter search for KNLMS was presented which achieves higher performance than any previously reported design. Pipeline stages are filled with multiple independent tasks, corresponding to different machine learning parameter values, allowing high utilisation of resources. This work demonstrated the feasibility of performing parameter search at nanosecond periods and opens the way for Big Data applications which were previously computationally intractable. Our PCI system achieves 9.9 / 2.6× speedup over a CPU / GPU implementation respectively.

Future work will focus on techniques to further increase parallelism, explore precision tradeoffs, reduce the latency of the design and to further explore how to accelerate larger problem sizes. Another possible use case for this design is to process high-bandwidth, independent streams of data. The system can process 8×32 bits continuously at 250 MHz, equating to a throughput of 64 Gbps.

Chapter 5

The Delayed Kernel Normalised Least Mean Squares Algorithm and Architectures

5.1 Introduction

In recent years, several online kernel-based adaptive filters have been proposed to solve non-linear regression problems. These KAFs utilise the Mercer kernel (Aronszajn, 1950) to allow linear techniques to be applied to non-linear feature spaces without directly computing the feature vectors (Cortes and Vapnik, 1995). Most KAFs are conceptually similar to their linear counterparts, e.g., the LMS and RLS algorithms, but have been reformulated in the *dual space* (Liu, Príncipe, and Haykin, 2011) and utilise Mercer kernels. KAFs form a growing research field related to machine learning and digital signal processing, particularly in real-time environments. KAFs are popular in applications that require: non-linear modelling capability; adaptation to changing conditions; and less computational demand than deep neural networks.

The potential benefits of KAFs have been demonstrated by several previous works including applications, such as channel equalisation (Van Vaerenbergh, Via, and Santamaria, 2006) and time series prediction (Richard, Bermudez, and Honeine, 2009), where high data rates may be required. However very few works, with the notable exception of Tridgell et al. (2015), have considered whether high throughput implementations are feasible. All KAFs are online algorithms with recursive expressions in order to update their models based on new examples. These create dependencies which must be addressed in order for high performance hardware to be created.

This chapter addresses this problem by considering the effect of pipelining the KNLMS algorithm. We show that by modifying the KNLMS algorithm to have *delayed model adaptation* (described in Section 5.3), high frequency implementations can be realised using FPGA technology. Specifically, the contributions of this chapter are:

- a technique for modifying KAFs, delayed model adaptation, to allow them to be pipelined;
- an application of delayed model adaptation to the KNLMS algorithm, producing the DKNLMS algorithm and further generalisation to multiple delays MDKNLMS;
- two further variations of DKNLMS, DKNLMS-DG and DKNLMS-CT, the latter of which can achieve higher accuracy while maintaining high performance;
- a description of hardware architectures to implement the KNLMS and DKNLMS algorithms with comparisons of speed, area and scalability;

- an empirical analysis of the learning accuracy of the DKNLMS algorithm including comparisons with LMS, KNLMS and other KAFs;
- an empirical analysis of the effect of using 18-bit fixed point precision and some function approximation (exponential, division) on the overall accuracy and stability of DKNLMS; and
- all source code used to generate the results in Section 5.6 has been open sourced,[1] so that interested readers may easily reproduce and extend the work.

The delayed model adaptation used in this work, can be considered as an extension to similar work on pipelining linear adaptive filters. In particular, the work by Long, Ling, and Proakis (1989), Poltmann (1995), Douglas, Zhu, and Smith (1998) and Yi et al. (2005). The architecture described in this work, can be considered as an extension to the work by Fraser et al. (2015a) to implement the DKNLMS variants.

The chapter is arranged as follows: Section 5.2 briefly explains kernel methods, the KNLMS algorithm and previous pipelined adaptive filtering architectures; Section 5.3 describes the delayed model adaptation method and the application of it to the KNLMS algorithm; Section 5.4 and 5.5 describe the proposed architecture and implementation details respectively; Section 5.6 shows accuracy results of the algorithm and the performance of the implementation; and finally, conclusions are drawn in Section 5.7.

5.2 Background

The KNLMS algorithm is a type of KAF (Liu, Príncipe, and Haykin, 2011), which are variants of adaptive filters (Haykin, 2005), such as LMS and RLS-based algorithms (Widrow and Hoff, 1960; Ding, 2013; Ding et al., 2018), utilising the kernel trick (Cortes and Vapnik, 1995). In this section, the KNLMS algorithm (Richard, Bermudez, and Honeine, 2009) is described and summarised. Issues associated with hardware implementations are highlighted including: dictionary growth and data dependencies. Previous implementations of KAFs and high performance linear adaptive filters are also briefly reviewed. Note, in this section we do a recap of some information from Sections 2.1 and 4.2. In particular, we recap some information about KNLMS from Section 4.2, but with slightly different notation to allow for simpler descriptions of the algorithms proposed in Section 5.3. If readers wish to skip the description of KNLMS, they should take note of the slight changes in notation before moving to Section 5.3.

5.2.1 Kernel Adaptive Filtering

KAFs are online algorithms which create non-linear models to fit a set of training examples. The training examples consist of input/output pairs $\mathbf{x}_n \in \mathbb{R}^M$ and $y_n \in \mathbb{R}$, representing the input vector and desired output value. The kernel adaptive filter model is represented by the following:

- A positive definite kernel function, $\kappa(\mathbf{x}_i, \mathbf{x}_j)$, chosen at design time. Examples of kernel functions are: the Gaussian kernel, $\kappa(\mathbf{x}_i, \mathbf{x}_j) = e^{-\gamma\|\mathbf{x}_i - \mathbf{x}_j\|_2^2}$, where γ is a parameter chosen at design time; the polynomial kernel, $\kappa(\mathbf{x}_i, \mathbf{x}_j) = \left(\mathbf{x}_i^T\mathbf{x}_j + c\right)^d$, where c and d are chosen at design time; and the linear kernel, $\kappa(\mathbf{x}_i, \mathbf{x}_j) = \mathbf{x}_i^T\mathbf{x}_j$.

[1] `https://bitbucket.org/nick_fraser/knlms_core_gen`

- A dictionary, given by $\mathcal{D}$, which is a subset of training example inputs.
- A vector of weights, $\boldsymbol{\alpha}$, where one weight corresponds to a single dictionary entry.

Given a new example, $\mathbf{x}_n$, a prediction, $\tilde{y}_n$, is calculated as follows:

$$\tilde{y}_n = \sum_{i=1}^{\tilde{N}_{n-1}} \alpha_i \kappa(\mathbf{x}_n, \tilde{\mathbf{x}}_i) \quad , \tag{5.1}$$

where $\tilde{N}_{n-1}$ is the number of entries in $\mathcal{D}$ at time $n-1$, henceforth denoted as $\mathcal{D}_{n-1}$, $\tilde{\mathbf{x}}_i$ is the i^{th} entry of $\mathcal{D}$, and α_i is the i^{th} entry of $\boldsymbol{\alpha}$. In general, the goal of KAFs is to find $\boldsymbol{\alpha}$ and $\mathcal{D}$ which accurately predicts $y_n \; \forall \; n$.

5.2.2 Kernel Normalised Least Mean Squares

Given a new example, $\{\mathbf{x}_n, y_n\}$, the goal of training is to update the model, $(\mathcal{D}_{n-1}, \hat{\boldsymbol{\alpha}}_{n-1})$, from time $n-1$ to time n, where $\hat{\boldsymbol{\alpha}}_{n-1}$ is the approximation to $\boldsymbol{\alpha}$ at time $n-1$.

In order to calculate the prediction of y_n and to evaluate the *coherence criterion* (Richard, Bermudez, and Honeine, 2009) (which decides whether to add $\mathbf{x}_n$ to the dictionary), a kernel evaluation must be made between $\mathbf{x}_n$ and all entries in the dictionary. We first define $\mathbf{k}_{n-1}(\cdot) \in \mathbb{R}^M \to \mathbb{R}^{\tilde{N}_{n-1}}$ as a function which, when applied to a new input, $\mathbf{x}_n$ is given by:

$$\mathbf{k}_{n-1}(\mathbf{x}_n) = [\kappa(\mathbf{x}_n, \tilde{\mathbf{x}}_1), \ldots, \kappa(\mathbf{x}_n, \tilde{\mathbf{x}}_{\tilde{N}_{n-1}})]^T \quad , \tag{5.2}$$

Conceptually, this kernel vector can be thought of as a vector of kernel evaluations between input $\mathbf{x}_n$ and each entry of $\mathcal{D}_{n-1}$. The coherence between $\mathbf{x}_n$ and $\mathcal{D}_{n-1}$ is given by:

$$\mu = \max_i \frac{|\kappa(\mathbf{x}_n, \tilde{\mathbf{x}}_i)|}{\sqrt{\kappa(\mathbf{x}_n, \mathbf{x}_n)\kappa(\tilde{\mathbf{x}}_i, \tilde{\mathbf{x}}_i)}} \quad \text{s.t. } \; i \in \left\{1, \cdots, \tilde{N}_{n-1}\right\} \quad . \tag{5.3}$$

The example, $\mathbf{x}_n$ is added to the dictionary if $\mu < \mu_0$, where μ_0 is a parameter specified at design time to control the size and coherence of the dictionary. The idea of the coherence criterion is to prevent redundant training examples from being added to the dictionary, i.e., if $\mathcal{D}_{n-1}$ contains an entry very similar to $\mathbf{x}_n$, then there would be limited benefit to adding $\mathbf{x}_n$ to $\mathcal{D}_{n-1}$. However, adding $\mathbf{x}_n$ would cost us more in memory and computational requirements of KNLMS at time n, so if $\mu \geq \mu_0$, then we prefer not to add $\mathbf{x}_n$ to the dictionary. Note that if $\kappa(\cdot,\cdot)$ is a unit norm kernel, Equation (5.3) can be simplified to:

$$\mu = \max_i |\kappa(\mathbf{x}_n, \tilde{\mathbf{x}}_i)| \quad \text{s.t. } \; i \in \left\{1, \cdots, \tilde{N}_{n-1}\right\} \quad . \tag{5.4}$$

The Gaussian kernel used in this work is a unit norm kernel, so we use Equation (5.4), rather than Equation (5.3).

The kernel vector is then updated, creating $\mathbf{k}_n(\mathbf{x}_n) = [\mathbf{k}_{n-1}(\mathbf{x}_n)^T, \kappa(\mathbf{x}_n, \mathbf{x}_n)]^T$ if $\mathbf{x}_n$ is added, or else simply $\mathbf{k}_n(\mathbf{x}_n) = \mathbf{k}_{n-1}(\mathbf{x}_n)$. Similarly, $\hat{\boldsymbol{\alpha}}_{n-1}$ is appended with a zero, if $\mathbf{x}_n$ is added to the dictionary. The KNLMS update step is derived by solving the following optimisation problem:

$$\hat{\boldsymbol{\alpha}}_n = \operatorname*{argmin}_{\boldsymbol{\alpha}} \|\boldsymbol{\alpha} - \hat{\boldsymbol{\alpha}}_{n-1}\|_2 \text{ s.t. } y_n = \mathbf{k}_n(\mathbf{x}_n)^T \boldsymbol{\alpha} \quad . \tag{5.5}$$

The solution to which can be found by minimising the following Lagrangian function:

$$J(\boldsymbol{\alpha}, \boldsymbol{\lambda}) = \|\boldsymbol{\alpha} - \hat{\boldsymbol{\alpha}}_n\|_2 + \lambda(y_n - \mathbf{k}_n(\mathbf{x}_n)^T \boldsymbol{\alpha}) \quad . \tag{5.6}$$

where λ is a Lagrangian multiplier. Differentiating Equation (5.6) with respect to $\boldsymbol{\alpha}$ and λ and solving for zero, results in the following expressions for $\hat{\boldsymbol{\alpha}}_n$:

$$2(\hat{\boldsymbol{\alpha}}_n - \hat{\boldsymbol{\alpha}}_{n-1}) = \lambda \mathbf{k}_n(\mathbf{x}_n) \tag{5.7}$$

$$\mathbf{k}_n(\mathbf{x}_n)^T \hat{\boldsymbol{\alpha}}_n = y_n \quad . \tag{5.8}$$

Solving these equations for $\hat{\boldsymbol{\alpha}}_n$ leads to following update equation:

$$\hat{\boldsymbol{\alpha}}_n = \hat{\boldsymbol{\alpha}}_{n-1} + \eta \frac{y_n - \mathbf{k}_n(\mathbf{x}_n)^T \hat{\boldsymbol{\alpha}}_{n-1}}{\epsilon + \|\mathbf{k}_n(\mathbf{x}_n)\|_2^2} \mathbf{k}_n(\mathbf{x}_n) \quad , \tag{5.9}$$

where η is a step size parameter and ϵ is a regularisation parameter, both of which are chosen at design time.

The KNLMS algorithm can be described using the psuedocode given in Algorithm 5.1. Note, that κ is assumed to be a unit norm kernel function.

Algorithm 5.1 KNLMS algorithm with coherence criterion and a unit norm kernel function.

```
Choose values for the step-size, $\eta$, and the regularisation factor, $\epsilon$.
Initialise $\mathcal{D} = \{\tilde{\mathbf{x}}_1\}$, $\boldsymbol{\alpha}_1 = [\frac{\eta}{1+\epsilon} y_1]$, $\tilde{N}_0 = 0$
while $n > 1$ do
    Get $\{\mathbf{x}_n, y_n\}$.
    Calculate $\mathbf{k}_{n-1}(\mathbf{x}_n) = [\kappa(\mathbf{x}_n, \tilde{\mathbf{x}}_1), \cdots, \kappa(\mathbf{x}_n, \tilde{\mathbf{x}}_{\tilde{N}_{n-1}})]^T$.
    $\mu = \max(|\mathbf{k}_{n-1}(\mathbf{x}_n)|)$.
    if $\mu < \mu_0$ then
        $\tilde{N}_n = \tilde{N}_{n-1} + 1$.
        Append $\mathbf{x}_n$ to $\mathcal{D}$.
        Append $0$ to $\hat{\boldsymbol{\alpha}}_{n-1}$.
        $\mathbf{k}_n(\mathbf{x}_n) = [\mathbf{k}_{n-1}(\mathbf{x}_n)^T, \kappa(\mathbf{x}_n, \mathbf{x}_n)]^T$.
    else
        $\tilde{N}_n = \tilde{N}_{n-1}$.
        $\mathbf{k}_n(\mathbf{x}_n) = \mathbf{k}_{n-1}(\mathbf{x}_n)$.
    end if
    Calculate $\hat{\boldsymbol{\alpha}}_n$ using Equation (5.9).
end while
```

5.2.3 KNLMS In Hardware

In order to make the KNLMS algorithm more amenable to high-performance hardware implementations, the following modifications are made:

- a maximum dictionary size is chosen, $\tilde{N}$;
- the kernel vector is the same length, $\tilde{N}$, at each iteration - if the dictionary is not full, the unused entries are padded with zeros; and
- similarly, the weights remain the same length, $\tilde{N}$, at each iteration - since the kernel vector is padded with zeros, unused weights will remain zero.

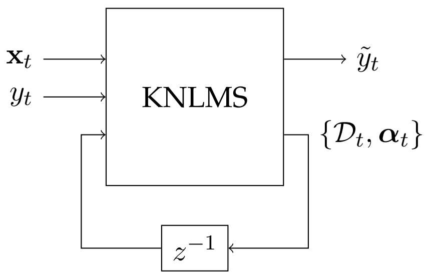


FIGURE 5.1: An illustration of the bottleneck in KAFs.

If $\tilde{N}$ is larger than the number of entries that would be allowed into the dictionary due to the coherence criterion, then KNLMS will perform exactly the same with or without the above modifications. The restriction on the maximum dictionary size means we can ensure that $\mathcal{D}$ and $\boldsymbol{\alpha}$ can always fit in on-chip memory, avoiding potential bottlenecks reading and writing to off-chip memory. Furthermore, this places a restriction on the maximum number of operations that are required per training example, which allows us to design hardware to meet minimum throughput requirements of a given target application. The padding of the kernel and weight vectors to be of size $\tilde{N}$ mean that all vector sizes in every iteration are the same, regardless of the number of entries in $\mathcal{D}$. This allows us to simplify our datapath design for fixed vector sizes and reduce control logic overhead.

5.2.4 Pipelined Adaptive Filters

The recursive nature of adaptive filters means that they suffer from a computational bottleneck, as illustrated in Figure 5.1 for KNLMS. In order to start calculating the model at time n, we must know the model at time $n-1$. Typically, this means that our data rate is limited by the latency of the critical path of the architecture for a single update step.

For linear adaptive filters, such as LMS, this problem has been considered in great detail. Long, Ling, and Proakis (1989) introduced the first analysis of the LMS algorithm with delayed coefficient adaptation, i.e., the delayed least mean squares (DLMS) algorithm. Yi et al. (2005) proposed variants of the DLMS algorithm together with implementation architectures. In particular, their transpose-form delayed least mean squares (TF-DLMS) implementation utilises a pipeline with different delay factors depending on the spatial location of filter weights. The architectures proposed by Yi et al. (2005) implement delayed LMS, TF-DLMS and some combinations in-between along with analysis of operation counts and expected propagation delay. Furthermore, Yi et al. (2005) then systematically compares these algorithms and architectures and is able to achieve data rates up to 182 MS/s on a Virtex-II FPGA for a variant they propose called transpose-form retimed delayed least mean squares (TF-RDLMS). Douglas, Zhu, and Smith (1998) describe an architecture which utilises error correction terms, proposed by Poltmann (1995), to create a pipelined LMS architecture which has the same behaviour as the standard LMS algorithm. The error correction terms allow the weights to be updated after some latency without diverging from the original algorithm.

To our knowledge, no prior work has studied the effects of model adaptation delay on KAFs. However, there are several works which propose architectures for executing various types of KAFs. Ren et al. (2014) propose an architecture for the KLMS algorithm which is designed to consume a small amount of resources while using floating point arithmetic. A high degree of parallelism is achieved by replicating the design across an FPGA device. The design also utilised a very efficient kernel, the survival kernel (Chen, Zheng, and Principe, 2013), which can achieve high accuracy but is not as common as the Gaussian kernel used in this work.

Fraser et al. (2017a) describe a deeply pipelined processor generator which can produce dataflow implementations of the KNLMS algorithm. Architecturally, it is similar to this work. The authors provide resource and performance results for the architecture using both floating point and fixed point arithmetic. However, the architecture and implementation described by Fraser et al. (2017a) requires deep pipelines which would result in very high delay factors, if the hardware were repurposed for implementing one of the DKNLMS-based algorithms described in this work. Despite having vastly different architectures, the designs of both Ren et al. (2014) and Fraser et al. (2017a) achieve parallelism across hyperparameter search and/or multi-channel systems but are not optimised for the single-channel, high-throughput case. The present work directly addresses this problem.

Pang et al. (2016) describe a pipelined, microcoded vector processor. The processor is compact, power efficient and capable of achieving high performance on SW-KRLS (Van Vaerenbergh, Via, and Santamaria, 2006), FB-KRLS (Van Vaerenbergh et al., 2010) and KNLMS (Richard, Bermudez, and Honeine, 2009). The processor used floating point arithmetic, and is one of the few KAFs architectures capable of handling multiple algorithm types.

Fox, Boland, and Leong (2018) describe an architecture which implements the Fastfood algorithm (Le, Sarlós, and Smola, 2013) which approximates $\mathbf{k}_{n-1}(\mathbf{x}_n)$ using random projections. The matricies used to compute $\mathbf{k}_{n-1}(\mathbf{x}_n)$ can be defined in a way which constrains them to be Hadamard matricies, allowing the most computationally expensive part of the update step to be computed using the fast Walsh-Hadamard transform (FWHT). This reduces the computational complexity from $O(\tilde{N}M) \rightarrow O(\tilde{N}\log_2 M)$. A systolic structure of Hadamard blocks, with an internal butterfly structure to efficiently implement the FWHT was utilised for its implementation. The work targeted larger scale problems, which means they can achieve high Processing Element (PE) efficiency, but lower sample rates, compared to this work.

Tridgell et al. (2015) describe the *braiding* technique to handle dependencies in high performance KAFs. To minimise latency, braiding calculates both branches of a conditional statement in parallel, and inserts the result of the appropriate branch into a reduction tree once the condition is known. The authors demonstrate the technique with a high throughput implementation of naive online regularised risk minimisation algorithm (NORMA) (Kivinen, Smola, and Williamson, 2004), utilising fixed point arithmetic. Their provided post-place-and-route results demonstrate cores capable of achieving a throughput of 138 MS/s. The design suffers from only being able to work on sliding window algorithms. In contrast, this work can be applied to arbitrary dictionary storage schemes.

5.3 Kernel Normalised Least Mean Squares With Delayed Model Adaptation

In this section, we provide a mathematical description of the DKNLMS algorithm. This has KNLMS-like behaviour, with dependencies shifted temporally into the future to facilitate pipelining. Motivations behind these choices are further elaborated upon in the Architecture section (Section 5.4).

The DKNLMS algorithm is conceptually similar to the delayed LMS algorithm (Long, Ling, and Proakis, 1989). The key idea is to introduce a delay factor, $d \in \mathbb{N}^{+}$, which

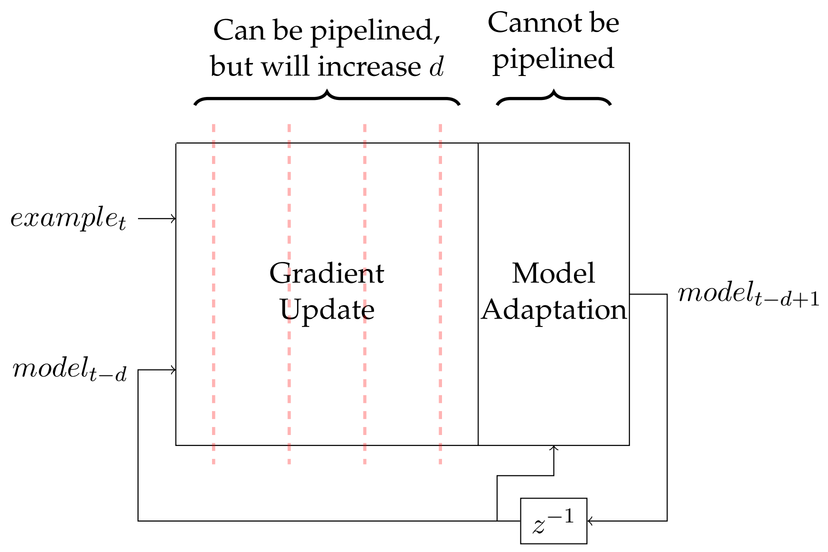


FIGURE 5.2: Pipelining the gradient calculation increases d, but will increase the sample rate. Pipelining the model adaptation will increase the number of channels while decreasing the sample rate.

removes most of the dependencies between iterations up to time d. At time n, given a new training pair, $\{\mathbf{x}_n, y_n\}$, and a delayed version of the model, given by $\mathcal{D}_{n-d}$ and $\tilde{\boldsymbol{\alpha}}_{n-d}$, we wish to calculate $\mathcal{D}_n$ and $\tilde{\boldsymbol{\alpha}}_n$. This is achieved by first, conceptually breaking the KNLMS algorithm up into two components:

1. the *gradient* calculation; and
2. the *model adaptation*.

The gradient calculation is the most computational intensive part of the KNLMS algorithm, whereas the model adaptation does not necessarily have long critical paths. Specifically, we consider the gradient calculation to involve: the calculation of $\mathbf{k}$, the coherence criterion, the a-priori prediction and the change in weights. The model adaptation is the storing of $\mathbf{x}_n$ into $\mathcal{D}_{n-d}$ (if required) and the accumulation of weights. If we design our dataflow so that that the algorithmic loop is contained to the model adaptation, then we can pipeline the gradient calculation arbitrarily, with each pipeline stage increasing d by 1. Figure 5.2, illustrates this mapping of the dataflow. Note that the recursive loop is coupled to the model adaptation, allowing the gradient calculation to be pipelined.

The kernel vector, $\mathbf{k}_{n-d}(\mathbf{x}_n)$, is calculated for the new input example. Similarly, the coherence between $\mathcal{D}_{n-d}$ and $\mathbf{x}_n$ is used to decide whether or not to add $\mathbf{x}_n$ to $\mathcal{D}_{n-1}$.

In order to update the weights, we need a prediction of y_n using the model at time $n-d$. This can be expressed as:

$$\hat{y}_n = \mathbf{k}_{n-d}(\mathbf{x}_n)^T \hat{\boldsymbol{\alpha}}_{n-d} \ . \tag{5.10}$$

A partially updated kernel vector, $\tilde{\mathbf{k}}_n(\mathbf{x}_n)$, if $\mathbf{x}_n$ is added to dictionary is given by:

$$\tilde{\mathbf{k}}_n(\mathbf{x}_n) = \left[\mathbf{k}_{n-d}(\mathbf{x}_n)^T, \mathbf{0}^T, \kappa(\mathbf{x}_n, \mathbf{x}_n)\right]^T \ , \tag{5.11}$$

where $\mathbf{0}$ is an $(\tilde{N}_{n-1} - \tilde{N}_{n-d})$ vector of zeros to align the latest dictionary entry with the corresponding entry in $\tilde{\boldsymbol{\alpha}}_n$. Alternatively, if $\mathbf{x}_n$ is not added then $\tilde{\mathbf{k}}_n(\mathbf{x}_n) =$

$\left[\mathbf{k}_{n-d}(\mathbf{x}_n)^T, \mathbf{0}^T\right]^T$. Finally, $\tilde{\boldsymbol{\alpha}}_n$ is calculated as follows:

$$\tilde{\boldsymbol{\alpha}}_n = \tilde{\boldsymbol{\alpha}}_{n-1} + \eta \frac{y_n - \hat{y}_n}{\epsilon + \left\| \tilde{\mathbf{k}}_n(\mathbf{x}_n) \right\|_2^2} \tilde{\mathbf{k}}_n(\mathbf{x}_n) \ , \tag{5.12}$$

Note that if $d = 1$, then Equations (5.10) to (5.12) are equivalent to KNLMS, i.e., Equations (5.1), (5.2) and (5.9). As such, DKNLMS can be considered as a generalisation to KNLMS to support $d > 1$.

The DKNLMS algorithm is described as psuedocode in Algorithm 5.2. Note that $\mathcal{D}_i = [\,]$, and $\tilde{\boldsymbol{\alpha}}_i = [\,]$ if $i \in \mathbb{Z}^-$ and a unit norm kernel is assumed for the coherence calculation. Also note, that we define $\mathcal{D}_n = \mathcal{D}_0$, $\tilde{\boldsymbol{\alpha}}_n = \tilde{\boldsymbol{\alpha}}_0$ and $\tilde{N}_n = \tilde{N}_0$, when $n < 0$.

Algorithm 5.2 DKNLMS algorithm with coherence criterion.

```
Initialise the step-size, $\eta$, and the regularisation factor, $\epsilon$, and delay factor, $d$.
Define a kernel function, $\kappa(\cdot, \cdot)$.
Initialise $\mathcal{D}_0 = [\,]$, $\tilde{\boldsymbol{\alpha}}_0 = [\,]$, and $\tilde{N}_0 = 0$.
while $n > 0$ do
    Get $\{\mathbf{x}_n, y_n\}$.
    if $\tilde{N}_{n-d} == 0$ then
        $\mathbf{k}_{n-d}(\mathbf{x}_n) = [\,]$.
        $\mu = 0$.
    else
        Calculate $\mathbf{k}_{n-d}(\mathbf{x}_n) = [\kappa(\mathbf{x}_n, \tilde{\mathbf{x}}_1), \cdots, \kappa(\mathbf{x}_n, \tilde{\mathbf{x}}_{\tilde{N}_{n-d}})]^T$.
        $\mu = \max(|\mathbf{k}_{n-d}(\mathbf{x}_n)|)$.
    end if
    if $\mu < \mu_0$ then
        $\tilde{N}_n = \tilde{N}_{n-1} + 1$.
        $\mathcal{D}_n = [\mathcal{D}_{n-1}; \mathbf{x}_n]$.
        Append $0$ to $\tilde{\boldsymbol{\alpha}}_{n-1}$.
        $\tilde{\mathbf{k}}_n(\mathbf{x}_n) = [\mathbf{k}_{n-d}(\mathbf{x}_n)^T, \mathbf{0}^T, \kappa(\mathbf{x}_n, \mathbf{x}_n)]^T$.
    else
        $\tilde{N}_n = \tilde{N}_{n-1}$.
        $\mathcal{D}_n = \mathcal{D}_{n-1}$.
        $\tilde{\mathbf{k}}_n(\mathbf{x}_n) = \mathbf{k}_{n-d}(\mathbf{x}_n)$.
    end if
    Calculate $\tilde{\boldsymbol{\alpha}}_n$ using Equation (5.12).
end while
```

5.3.1 Multiple Delays

When studying the dataflow in Algorithm 5.2 it is clear that the values for the dictionary delay, $d_{\mathcal{D}} \in \mathbb{N}^+$, and the weight delay, $d_{\boldsymbol{\alpha}} \in \mathbb{N}^+$, need not be the same. A version of DKNLMS with multiple delays, which is henceforth referred to as the MDKNLMS algorithm, can easily be implemented in hardware by splitting the DKNLMS into the two parts shown in Figure 5.3. This modification incurs no further computation cost, and actually reduces the number of registers required within the design, which may reduce the resource usage of the design. Similar to Figure 5.2, each hardware block shown in Figure 5.3 can be pipelined, resulting in specific values for $d_{\mathcal{D}} \in \mathbb{N}^+$ and $d_{\boldsymbol{\alpha}} \in \mathbb{N}^+$. It is unclear as to whether this change in delay of the algorithm's dependencies will have a positive or negative affect on the maximum

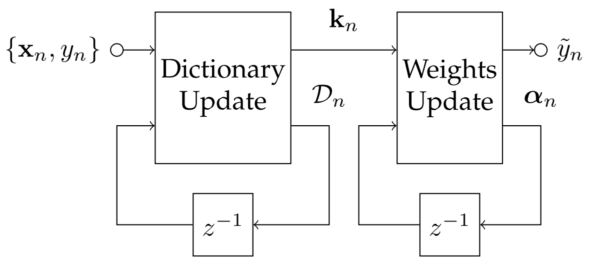


FIGURE 5.3: A high level diagram of MDKNLMS.

clock frequency of an implementation, we analyse this empirically in Section 5.6. Comparing the DKNLMS and MDKNLMS algorithms, for the same number of pipeline stages, $N_p = d$, MDKNLMS will reduce the effective delay of either $d_{\mathcal{D}}$, $d_{\boldsymbol{\alpha}}$, or both. That is, for DKNLMS the delay factors are given by $N_p = d = d_{\mathcal{D}} = d_{\boldsymbol{\alpha}}$, while for MDKNLMS are $N_p = d = d_{\mathcal{D}} + d_{\boldsymbol{\alpha}}$. Furthermore, if a higher model adaptation delay introduces an instability in the algorithm, as can occur for delayed LMS (Long, Ling, and Proakis, 1989), then MDKNLMS will be more stable than the equivalent DKNLMS with an equivalent delay.

Explicitly for MDKNLMS, the a-priori prediction, $\bar{y}$, becomes:

$$\bar{y}_n = \mathbf{k}_{n-d_{\mathcal{D}}}(\mathbf{x}_n)^T \bar{\boldsymbol{\alpha}}_{n-d_{\boldsymbol{\alpha}}} \quad , \tag{5.13}$$

where either $\mathbf{k}_{n-d_{\mathcal{D}}}(\mathbf{x}_n)$ or $\bar{\boldsymbol{\alpha}}_{n-d_{\boldsymbol{\alpha}}}$ are appended with zeros to accommodate any size mismatches, $\bar{\boldsymbol{\alpha}}_{n-d_{\boldsymbol{\alpha}}}$ is given by:

$$\bar{\boldsymbol{\alpha}}_n = \bar{\boldsymbol{\alpha}}_{n-1} + \eta \frac{y_n - \bar{y}_n}{\epsilon + \left\|\bar{\mathbf{k}}_n(\mathbf{x}_n)\right\|_2^2} \bar{\mathbf{k}}_n(\mathbf{x}_n) \quad , \tag{5.14}$$

where $\bar{\mathbf{k}}_n(\mathbf{x}_n)$ is the partially update kernel vector, given by:

$$\bar{\mathbf{k}}_n(\mathbf{x}_n) = \left[\mathbf{k}_{n-d_{\mathcal{D}}}(\mathbf{x}_n)^T, \mathbf{0}^T, \kappa(\mathbf{x}_n, \mathbf{x}_n)\right]^T \quad . \tag{5.15}$$

Pseudocode for the MDKNLMS algorithm is given in Algorithm 5.3. Note, that κ is assumed to be a unit norm kernel function. Comparing Algorithms 5.2 and 5.3 it is clear that the only difference between to two algorithms is that MDKNLMS uses $d_{\mathcal{D}}$ and $d_{\boldsymbol{\alpha}}$ while DKNLMS uses only d. If $N_p = d = d_{\mathcal{D}} + d_{\boldsymbol{\alpha}}$ then MDKNLMS should be less impacted algorithmically from the delay (when compared to KNLMS) than DKNLMS, as any delay introduced in update of its model parameters are less than or equal to those in DKNLMS. This claim is evaluated empirically in Section 5.6.

5.3.2 Dictionary Guarding

An issue with applying delayed model adaptation to KNLMS is that the choice of whether or not to add an example to $\mathcal{D}_{n-1}$ is based on an outdated dictionary, $\mathcal{D}_{n-d_{\mathcal{D}}}$. This can result in a dictionary which contains *redundant* entries. For example, imagine that two consecutive examples are equal, i.e., $\mathbf{x}_n = \mathbf{x}_{n+1}$ (the outputs y_n, y_{n+1} can be ignored for now), and the coherence between $\mathbf{x}_n$ and all earlier examples are less than μ_0. Clearly, when training occurs on $\mathbf{x}_n$, the coherence between it and $\mathcal{D}_{n-d_{\mathcal{D}}}$ will be less than μ_0, and it will be added to the dictionary, $\mathcal{D}_{n-1}$, to create $\mathcal{D}_n$. When training occurs on $\mathbf{x}_{n+1}$, the coherence between $\mathbf{x}_{n+1}$ and $\mathcal{D}_{n-d_{\mathcal{D}}+1}$ will also be less than μ_0. As such, $\mathbf{x}_{n+1}$ is also added to the dictionary, creating $\mathcal{D}_{n+1}$. Given that the prediction function, Equation (5.1), is based on the weighted sum of kernel evaluations between

Algorithm 5.3 MDKNLMS algorithm with coherence criterion.

```
Initialise the step-size, $\eta$, and the regularisation factor, $\epsilon$, dictionary delay, $d_{\mathcal{D}}$ and a
weight delay, $d_{\boldsymbol{\alpha}}$.
Define a kernel function, $\kappa(\cdot,\cdot)$.
Initialise $\mathcal{D}_0 = [\,]$, $\tilde{\boldsymbol{\alpha}}_0 = [\,]$, and $\tilde{N}_0 = 0$.
while $n > 0$ do
    Get $\{\mathbf{x}_n, y_n\}$.
    if $\tilde{N}_{n-d_{\mathcal{D}}} == 0$ then
        $\mathbf{k}_{n-d_{\mathcal{D}}}(\mathbf{x}_n) = [\,]$.
        $\mu = 0$.
    else
        Calculate $\mathbf{k}_{n-d_{\mathcal{D}}}(\mathbf{x}_n) = [\kappa(\mathbf{x}_n, \tilde{\mathbf{x}}_1), \cdots, \kappa(\mathbf{x}_n, \tilde{\mathbf{x}}_{\tilde{N}_{n-d}})]^T$.
        $\mu = \max(|\mathbf{k}_{n-d_{\mathcal{D}}}(\mathbf{x}_n)|)$.
    end if
    if $\mu < \mu_0$ then
        $\tilde{N}_n = \tilde{N}_{n-1}$.
        $\mathcal{D}_n = \mathcal{D}_{n-1}$.
        Append 0 to $\bar{\boldsymbol{\alpha}}_{n-1}$.
        $\bar{\mathbf{k}}_n(\mathbf{x}_n) = [\mathbf{k}_{n-d_{\mathcal{D}}}(\mathbf{x}_n)^T, \mathbf{0}^T, \kappa(\mathbf{x}_n, \mathbf{x}_n)]^T$.
    else
        $\tilde{N}_n = \tilde{N}_{n-1}$.
        $\mathcal{D}_n = \mathcal{D}_{n-1}$.
        $\bar{\mathbf{k}}_n(\mathbf{x}_n) = \mathbf{k}_{n-d_{\mathcal{D}}}(\mathbf{x}_n)$.
    end if
    Calculate $\bar{\boldsymbol{\alpha}}_n$ using Equation (5.14).
end while
```

the dictionary entries and the latest input, we can see that $\mathcal{D}_{n+1}$ provides no improved predictive power over $\mathcal{D}_n$, i.e., a prediction made using $\mathcal{D}_{n+1}$ could be rewritten as:

$$\sum_{i=1}^{\tilde{N}_{n+1}} \alpha_i \kappa(\mathbf{x}_j, \tilde{\mathbf{x}}_i) = \sum_{i=1}^{\tilde{N}} \alpha_i \kappa(\mathbf{x}_j, \tilde{\mathbf{x}}_i) + (\alpha_{\tilde{N}_n} + \alpha_{\tilde{N}_{n+1}}) \kappa(\mathbf{x}_j, \tilde{\mathbf{x}}_{\tilde{N}_n}) \ . \tag{5.16}$$

This means that $\mathcal{D}_{n+1}$ uses more memory and has a higher computational cost, for no improved modelling accuracy over $\mathcal{D}_n$.

To address this effect, we propose a method to prevent redundant entries from being added to the dictionary, which we call *dictionary guarding*. If $\mathbf{x}_n$ is added, based on the dictionary at time $n - d_{\mathcal{D}}$, it is not used to assist in making a decision of whether or not a new example should be added until $\mathbf{x}_{n+d_{\mathcal{D}}}$ arrives, i.e., we suggest, that the examples $\{\mathbf{x}_{n+1}, \cdots, \mathbf{x}_{n+d_{\mathcal{D}}-1}\}$ are prevented from being added to the dictionary.

In terms of implementation, the DKNLMS-DG is almost identical to MDKNLMS, with the exception of the decision to add an entry to the dictionary, which now based on the coherence criterion and a counter, c_n. Psuedocode for the DKNLMS-DG algorithm is given in Algorithm 5.4. Note, that κ is assumed to be a unit norm kernel function.

Although the changes made to MDKNLMS to achieve DKNLMS-DG are minimal, since KAFs are dynamic algorithms, the dictionary guarding technique may have a significant impact on the behaviour of the algorithm. As DKNLMS-DG introduces and extra requirement in order to introduce an entry to the dictionary, it will produce a dictionary that's more compact than MDKNLMS. However, this may also have the following negative side-effects:

- a decrease in modelling accuracy;
- an increase in convergence time; and
- a reduction in coherence between the dictionary and previously seen input examples, i.e., the coherence between $\mathcal{D}$ and all inputs is no longer guaranteed to be greater than μ_0.

How these issues relate to the resultant DKNLMS-DG algorithm are examined empirically in Section 5.6.

5.3.3 Correction Terms

If the accuracy of MDKNLMS is not satisfactory, we can modify it to exhibit equivalent learning behaviour to the original KNLMS algorithm. These modifications come at the cost of extra hardware, and potentially, a decrease in clock frequency. This section describes this modified version of MDKNLMS, DKNLMS-CT. These correction terms can be considered as an extension to the work by Poltmann (1995) to KAFs, a significant difference being that for KAFs the contents of the dictionary also need to be considered. These correction terms also share similarity to braiding (Tridgell et al., 2015), the difference being the correction terms described in this work can work for non-sliding window based KAFs.

Firstly, let us consider the dictionary update step. In KNLMS, the current example, $\mathbf{x}_n$, is added to the dictionary if $\mu < \mu_0$, where μ is calculated using Equation (5.4). The same applies to MDKNLMS, however, the calculation of μ is now based on an older version of the dictionary, $\mathcal{D}_{n-d_{\mathcal{D}}}$, rather than $\mathcal{D}_{n-1}$. At each time step, either the current input vector is added to the dictionary, or it remains unchanged. Therefore,

Algorithm 5.4 DKNLMS-DG algorithm with coherence criterion.

```
Initialise the step-size, $\eta$, and the regularisation factor, $\epsilon$, dictionary delay, $d_{\mathcal{D}}$ and a weight delay, $d_{\boldsymbol{\alpha}}$.
Define a kernel function, $\kappa(\cdot,\cdot)$.
Initialise $\mathcal{D}_0 = [\,]$, $\tilde{\boldsymbol{\alpha}}_0 = [\,]$, and $\tilde{N}_0 = 0$.
Initialise the dictionary guard counter $c_0 = 0$
while $n > 0$ do
    Get $\{\mathbf{x}_n, y_n\}$.
    if $\tilde{N}_{n-d_{\mathcal{D}}} == 0$ then
        $\mathbf{k}_{n-d_{\mathcal{D}}}(\mathbf{x}_n) = [\,]$.
        $\mu = 0$.
    else
        Calculate $\mathbf{k}_{n-d_{\mathcal{D}}}(\mathbf{x}_n) = [\kappa(\mathbf{x}_n, \tilde{\mathbf{x}}_1), \cdots, \kappa(\mathbf{x}_n, \tilde{\mathbf{x}}_{\tilde{N}_{n-d}})]^T$.
        $\mu = \max(|\mathbf{k}_{n-d_{\mathcal{D}}}(\mathbf{x}_n)|)$.
    end if
    if $(\mu < \mu_0)$ & $c_{n-1} == 0$ then
        $\tilde{c}_n = d_{\mathcal{D}} + d_{\boldsymbol{\alpha}}$
        $\tilde{N}_n = \tilde{N}_{n-1}$.
        $\mathcal{D}_n = \mathcal{D}_{n-1}$.
        Append 0 to $\bar{\boldsymbol{\alpha}}_{n-1}$.
        $\bar{\mathbf{k}}_n(\mathbf{x}_n) = [\mathbf{k}_{n-d_{\mathcal{D}}}(\mathbf{x}_n)^T, \mathbf{0}^T, \kappa(\mathbf{x}_n, \mathbf{x}_n)]^T$.
    else
        $\tilde{c}_n = c_{n-1}$
        $\tilde{N}_n = \tilde{N}_{n-1}$.
        $\mathcal{D}_n = \mathcal{D}_{n-1}$.
        $\bar{\mathbf{k}}_n(\mathbf{x}_n) = \mathbf{k}_{n-d_{\mathcal{D}}}(\mathbf{x}_n)$.
    end if
    Calculate $\bar{\boldsymbol{\alpha}}_n$ using Equation (5.14).
    if $\tilde{c}_n > 0$ then
        $c_n = \tilde{c}_n - 1$
    else
        $c_n = 0$
    end if
end while
```

$\mathcal{D}_{n-d_{\mathcal{D}}}$ is a subset of $\mathcal{D}_{n-1}$. $\mathcal{D}_{n-1}$ can also potentially contain any of the vectors from $\mathbf{x}_{n-d_{\mathcal{D}}+1} \rightarrow \mathbf{x}_{n-1}$. Given this, an expression for the decision value, ζ_n, the decision whether or not to add $\mathbf{x}_n$ to the dictionary, can be written as:

$$\zeta_n = ! \left[(\kappa(\mathbf{x}_n, \tilde{\mathbf{x}}_1) < \mu_0) \ \& \ \cdots \ \& \ (\kappa(\mathbf{x}_n, \tilde{\mathbf{x}}_{\tilde{N}_{n-1}}) < \mu_0) \right] \ , \tag{5.17}$$

where '&', '!' are the Boolean functions 'AND' and 'NOT' respectively. If ζ_n is true, then $\mathbf{x}_n$ is added to the dictionary. Note that unit norm kernel functions are assumed in Equation (5.17). If only $\mathcal{D}_{n-d_{\mathcal{D}}}$ is available, Equation (5.17) can be rewritten as:

$$\begin{aligned}\zeta_n =& ![(\kappa(\mathbf{x}_n, \tilde{\mathbf{x}}_1) < \mu_0) \ \& \ \cdots \ \& \ (\kappa(\mathbf{x}_n, \tilde{\mathbf{x}}_{\tilde{N}_{n-d_{\mathcal{D}}}}) < \mu_0) \\ & \& \ (\kappa(\mathbf{x}_n, \mathbf{x}_{n-d_{\mathcal{D}}+1}) < \mu_0 + !\zeta_{n-d_{\mathcal{D}}+1}) \ \& \ \cdots \ \& \ (\kappa(\mathbf{x}_n, \mathbf{x}_{n-1}) < \mu_0 + !\zeta_{n-1})] \ .\end{aligned} \tag{5.18}$$

Note that in this equation, '+' denotes the 'OR' function. While these expressions are equivalent, Equation (5.18) can be implemented easily using pipelined hardware and therefore allows tradeoffs between area usage and throughput to be made. Conceptually, Equation (5.18) can be though of as checking the coherence between $\mathbf{x}_n$, all dictionary entries and all training examples which are in the pipeline ahead of the current entry. The final decision is then corrected if any preceding training examples are added to the dictionary. This correction only takes place if the preceding entry is also coherent with the current training example, $\mathbf{x}_n$.

Now that the dictionary, $\mathcal{D}_n$, and kernel vector, $\mathbf{k}_n(\mathbf{x}_n)$, found by DKNLMS-CT are equivalent to KNLMS, we need to apply correction terms so that the weights, $\hat{\boldsymbol{\alpha}}$, are the same as those found by KNLMS. Let us consider Equations (5.9) and (5.12). Since the kernel vectors have been corrected, the only differences between these equations are the a-priori predictions. Defining the weight update step, δ_n, for KNLMS to be:

$$\delta_n = \frac{\eta \left(y_n - \mathbf{k}_n(\mathbf{x}_n)^T \boldsymbol{\alpha}_n \right)}{\epsilon + \|\mathbf{k}_n(\mathbf{x}_n)\|_2^2} \ , \tag{5.19}$$

and similarly, $\tilde{\delta}_n$ to be:

$$\tilde{\delta}_n = \frac{\eta \left(y_n - \mathbf{k}_n(\mathbf{x}_n)^T \boldsymbol{\alpha}_{n-d_{\boldsymbol{\alpha}}} \right)}{\epsilon + \|\mathbf{k}_n(\mathbf{x}_n)\|_2^2} \ , \tag{5.20}$$

for MDKNLMS with a corrected kernel vector. To create the DKNLMS-CT algorithm, we must find a way to modify Equation (5.20) to become Equation (5.19). In order to realise a benefit from pipelining, these modifications to Equation (5.20) should only depend on data available at sample $n - d_{\boldsymbol{\alpha}}$. Given this requirement, an expression for δ_n can be formed using Equations (5.9), (5.19) and (5.20):

$$\begin{aligned}\delta_n &= \frac{\eta}{\epsilon + \|\mathbf{k}_n(\mathbf{x}_n)\|_2^2} \left(y_n - \mathbf{k}_n(\mathbf{x}_n)^T [\boldsymbol{\alpha}_{n-d_{\boldsymbol{\alpha}}} + \sum_{i=1}^{d_{\boldsymbol{\alpha}}-1} \delta_{n-i} \mathbf{k}_{n-i}(\mathbf{x}_{n-i})] \right) \\ \delta_n &= \frac{\eta}{\epsilon + \|\mathbf{k}_n(\mathbf{x}_n)\|_2^2} \left(y_n - \mathbf{k}_n(\mathbf{x}_n)^T \boldsymbol{\alpha}_{n-d_{\boldsymbol{\alpha}}} - \sum_{i=1}^{d_{\boldsymbol{\alpha}}-1} \delta_{n-i} \mathbf{k}_n(\mathbf{x}_n)^T \mathbf{k}_{n-i}(\mathbf{x}_{n-i}) \right) \ .\end{aligned} \tag{5.21}$$

Using this expression for δ_n, we can rewrite Equation (5.9) as:

$$\hat{\boldsymbol{\alpha}}_n = \hat{\boldsymbol{\alpha}}_{n-1} + \delta_n \mathbf{k}_n(\mathbf{x}_n) \ . \tag{5.22}$$

Pseudocode for the DKNLMS-CT algorithm is given in Algorithm 5.5. Note that mathematically, this is same as Algorithm 4.1, but it's expressed in a different way to allow it to be implemented on a pipelined accelerator.

Algorithm 5.5 DKNLMS-CT algorithm with coherence criterion.

```
Initialise the step-size, $\eta$, and the regularisation factor, $\epsilon$, dictionary delay, $d_{\mathcal{D}}$ and a weight delay, $d_{\boldsymbol{\alpha}}$.
Define a kernel function, $\kappa(\cdot,\cdot)$.
Initialise $\mathcal{D}_0 = [\,]$, $\tilde{\boldsymbol{\alpha}}_0 = [\,]$, and $\tilde{N}_0 = 0$.
while $n > 0$ do
    Get $\{\mathbf{x}_n, y_n\}$.
    if $\tilde{N}_{n-1} == 0$ then
        $\mathbf{k}_{n-1}(\mathbf{x}_n) = [\,]$.
        $\zeta_n = 1$.
    else
        Calculate $\zeta_n$ using Equation (5.18).
        Get $\mathbf{k}_{n-1}(\mathbf{x}_n)$ by selecting the appropriate entries used when calculating $\zeta_n$.
    end if
    if $\zeta_n == 1$ then
        $\tilde{N}_n = \tilde{N}_{n-1} + 1$.
        $\mathcal{D}_n = [\mathcal{D}_{n-1}; \mathbf{x}_n]$.
        Append $0$ to $\hat{\boldsymbol{\alpha}}_{n-1}$.
        $\mathbf{k}_n(\mathbf{x}_n) = [\mathbf{k}_{n-1}(\mathbf{x}_n)^T, \kappa(\mathbf{x}_n, \mathbf{x}_n)]^T$.
    else
        $\tilde{N}_n = \tilde{N}_{n-1}$.
        $\mathcal{D}_n = \mathcal{D}_{n-1}$.
        $\mathbf{k}_n(\mathbf{x}_n) = \mathbf{k}_{n-1}(\mathbf{x}_n)$.
    end if
    Calculate $\hat{\boldsymbol{\alpha}}_n$ using Equation (5.22).
end while
```

Clearly, if $\hat{\boldsymbol{\alpha}}_n$ is updated using Equation (5.21) rather than Equation (5.9), then more operations are required. However, as we will see in Section 5.6, this way of computing the weights is preferable since we can still receive the benefits of pipelining the architecture. This is due to the fact that: (1) the vectors, $\{\mathbf{k}_{n-1}(\mathbf{x}_{n-1}), \cdots, \mathbf{k}_{n-d_{\boldsymbol{\alpha}}+1}(\mathbf{x}_{n-d_{\boldsymbol{\alpha}}+1})\}$, are available at time n; and (2) the only dependencies within the pipeline are scalar values, $\{\delta_{n-1}, \cdots, \delta_{n-d_{\boldsymbol{\alpha}}+1}\}$, which can be easily fed back where appropriate with only a small overhead.

The correction terms also do not need to be added completely. A subset of the correction terms in Equation (5.21) may be used instead. This can allow for further trade-offs to be made between area, accuracy and throughput.

Note that we only analyse the theoretical costs and benefits of DKNLMS-CT in the subsequent sections. As we will see in Section 5.6, the other DKNLMS variants achieve high accuracy and throughput and as such, empirically we deem that the addition of correction terms is not a requirement. However, this may change for future KAF algorithms and datasets, so this theoretical analysis could be useful for future works.

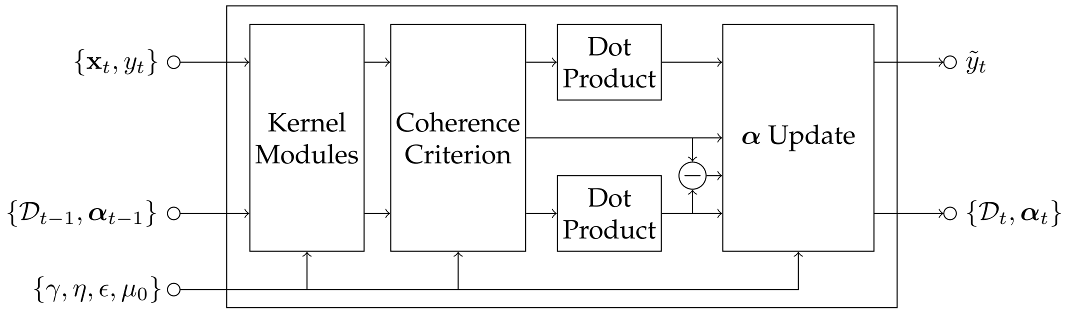

$\{\mathbf{x}_t, y_t\}$
$\{\mathcal{D}_{t-1}, \boldsymbol{\alpha}_{t-1}\}$
$\{\gamma, \eta, \epsilon, \mu_0\}$
Kernel Modules
Coherence Criterion
Dot Product
Dot Product
$\boldsymbol{\alpha}$ Update
$\tilde{y}_t$
$\{\mathcal{D}_t, \boldsymbol{\alpha}_t\}$

FIGURE 5.4: High level view of the proposed architecture.

5.4 Architecture

In this section, the architecture of the DKNLMS algorithm is described in detail. The variants MDKNLMS and DKNLMS-DG are simple extensions to DKNLMS, as such, we described the modifications where they should occur. The architecture for DKNLMS-CT is not described, rather we theoretically analyse the cost of implementing it. Finally, in this section we omit the subscripts denoting time to reduce the notation complexity.

5.4.1 High Level View

In order to understand the architecture at a high level, firstly consider the *forward path* of the KNLMS algorithm. The forward path consists of the operations required for the KNLMS algorithm to update its model (i.e., the contents of the while loop in Algorithm 5.1), given a new training example, $\{\mathbf{x}_n, y_n\}$. Since the forward path contains no loops, it can be implemented using combinatorial logic, which is later converted to implement DKNLMS. The design method can be summarised as follows:

1. begin with a combinatorial KNLMS design, created from a dataflow graph of the KNLMS forward path with a maximum dictionary size of $\tilde{N}$;
2. add registers (or memories) to store the dictionary and weights internally to the design (i.e., add the loops with the smallest possible critical path to Equation (5.12) and to the dictionary update step); and
3. excluding the loops described in (2), pipeline the rest of the dataflow graph until a desired d value (or clock frequency) is achieved.

A block diagram of the basic DKNLMS implementation is shown in Figure 5.4. The design can be thought of as the combination of four different submodules: kernel modules, the coherence criterion module, dot product modules and the $\boldsymbol{\alpha}$-update module. Note that the coherence criterion module and the $\boldsymbol{\alpha}$ update module contain loops and registers (or memories) to store and update the dictionary ($\mathcal{D}$) and weights ($\boldsymbol{\alpha}$) respectively. Throughout the design, 'optional' registers are placed between each arithmetic operation. Moreover, all arithmetic operations can also be pipelined while only affecting the delay parameters of the DKNLMS variants, with the exclusion of the accumulation on $\boldsymbol{\alpha}$.

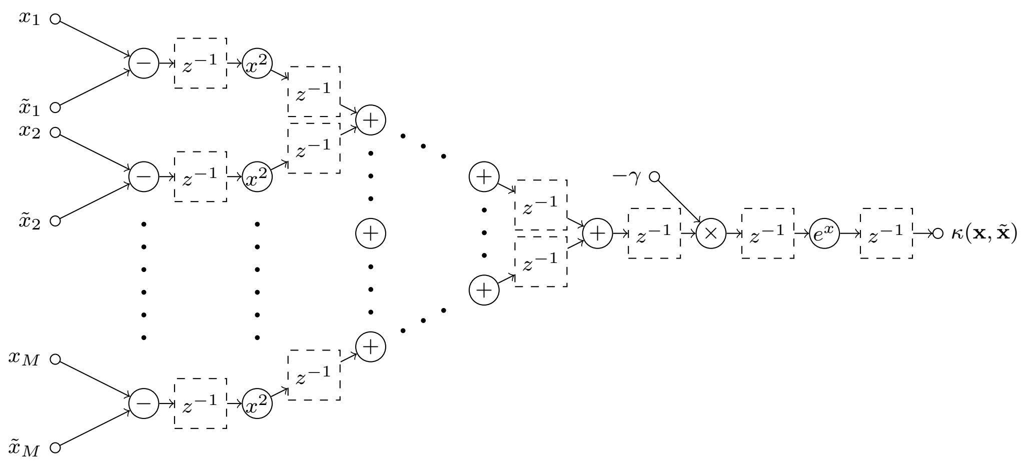


FIGURE 5.5: Exponential kernel evaluation module.

5.4.2 The Submodules

In this subsection, the submodules of the design (as shown in Figure 5.4) are explained in detail. The kernel modules form the largest portion of the architecture and are designed to calculate the kernel vector, $\mathbf{k}$, given in Equation (5.2). The ith element of the kernel vector is evaluated using the Gaussian kernel, given by: $\kappa(\mathbf{x}_n, \tilde{\mathbf{x}}_i) = \exp(\|\mathbf{x}_n - \tilde{\mathbf{x}}_i\|_2^2)$. Although there are several useful kernel functions described in the literature, including several which are hardware friendly (notably the Laplacian kernel, used by Anguita et al. (2007)), we chose to use the Gaussian kernel because: (1) it's a unit norm kernel; (2) it's a universal approximator (Hammer and Gersmann, 2003); and (3) it's a very popular kernel function, known to many familiar with kernel methods. In order to calculate $\mathbf{k}$, $\tilde{N}$ kernel modules are used, and Figure 5.5 illustrates the data path of a kernel module. Each kernel module takes the current input sample and an element of the dictionary as inputs, and produces the Gaussian kernel evaluation as an output. An adder tree is used to reduce latency. Note that we use z^{-1} with a broken line border to denote the optional registers. In order to use another unit norm kernel function, this module can simply be replaced with a module implementing your kernel function of choice. If a non-unit norm kernel is desired, the coherence module would also need to be modified to implement the coherence criterion described by Richard, Bermudez, and Honeine (2009).

The dot product module is given in Figure 5.6. It features elementwise multiplication followed by an adder tree. The dot product modules are used to calculate the a-priori prediction and $\|\mathbf{k}\|_2^2$, which is used to find the normalisation factor. For both the kernel and dot-product modules, an adder tree was preferred to perform accumulation, as we wish to minimise the overall latency.

The coherence criterion module calculates the coherence between the current input and the dictionary. The coherence (given in Equation (5.4)) is then used to determine whether or not to add the current input to the dictionary. Figure 5.7 shows the architecture of the coherence criterion module. It features comparison modules connected to a binary 'OR' tree, which produces the 'AddToDict' signal. This signal feeds a multiplexer which determines whether or not the current example gets added to the dictionary, and subsequently, whether or not the counter is incremented. The dictionary can be stored in registers or memories, but the current design requires that all dictionary entries must be read every cycle. This module also pads the kernel vector, $\mathbf{k}$, with zero entries for unused dictionary entries, and appends '1' to $\mathbf{k}$ if the

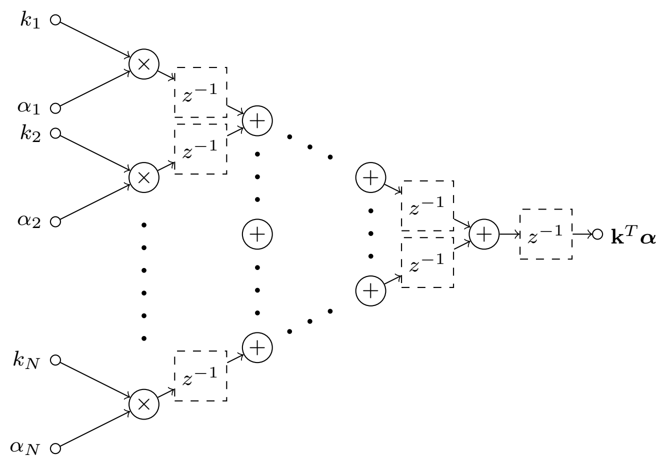


FIGURE 5.6: The dot product block was made using a simple binary tree structure.

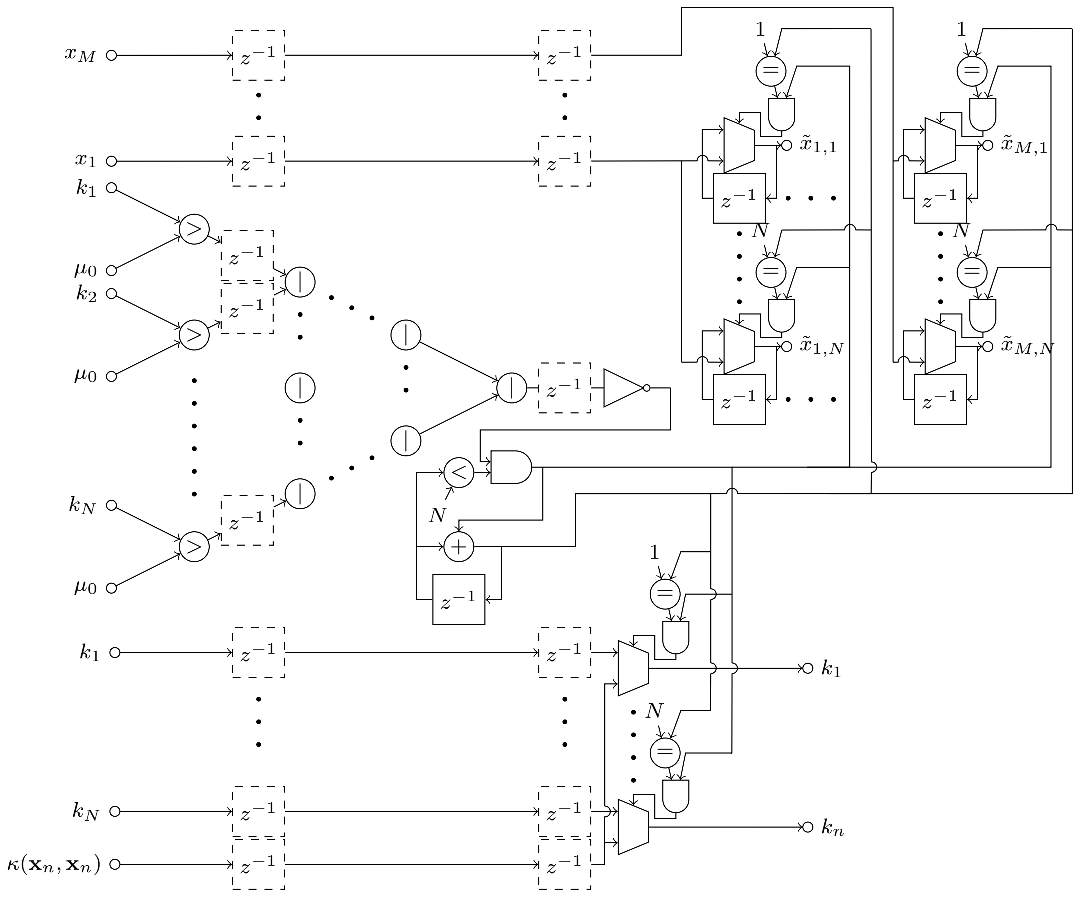


FIGURE 5.7: Coherence criterion module.

current input is added. In order to realise DKNLMS-DG, an extra counter must be included which value corresponds to the number of elements seen since the last entry was added to the dictionary. If this value is below $d_{\mathcal{D}}$, then 'AddToDict' is set to false.

Figure 5.8 shows the $\boldsymbol{\alpha}$ update module. This module takes the prediction error, e_n, the normalisation factor, $\|\mathbf{k}\|_2^2$, and the kernel vector, $\mathbf{k}$, as inputs. The output is the updated weights, which are also stored internally in registers. The $\boldsymbol{\alpha}$ update module calculates the weight update using Equation (5.9).

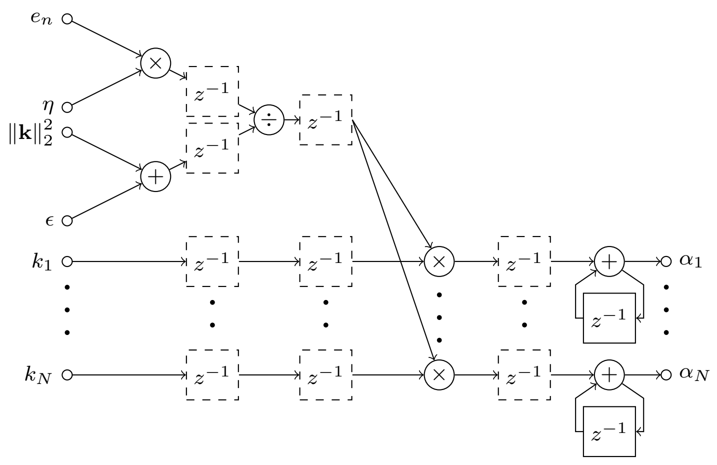


FIGURE 5.8: Incremental update of alpha.

TABLE 5.1: Resource usage of each module and the complete design

Module	Num. of Modules	exp	$\times$	$+$	$\div$	$<$
Kernel	$\tilde{N}$	1	$M+1$	$2M-1$	0	0
Dot Product	2	0	$\tilde{N}$	$\tilde{N}-1$	0	0
Coherence Criterion	1	0	0	0	0	$\tilde{N}$
α Update	1	0	$\tilde{N}+1$	$\tilde{N}+1$	1	0
Design Total	-	$\tilde{N}$	$(M+4)\tilde{N}+1$	$2(M+1)\tilde{N}$	1	$\tilde{N}$
Correction Terms ($d_{\mathcal{D}}$)	$d_{\mathcal{D}}-1$	1	$M+1$	$2M-1$	0	1
Correction Terms (d_{α})	$d_{\alpha}-1$	0	$\tilde{N}+1$	$\tilde{N}$	0	0

Both the coherence and α modules feature valid and reset signals. The valid signal prevents unwanted entries from being added to the dictionary and from modifying the weights. The reset signal restores the counter, dictionary and weights to zero values.

5.4.3 Resource Usage and Latency

In this subsection, the resource usage and propagation delay of the proposed architecture are given as a function of: (1) the maximum dictionary size, $\tilde{N}$; (2) the feature length, M; and (3) the operators, exp, $\times$, $+$, $/$ and $<$. Since the design is fully pipelined, modules do not share resources. Also, as shown in Figure 5.4, the four difference modules are connected in series. Therefore, the total latency of the design will be the sum of the latency of each of the four modules described above, plus a subtraction. The total resource usage of the design will be the sum of all resources of all modules used within the architecture, plus a subtraction.

By careful inspection of Figures 5.4 to 5.8 the total resource usage can be calculated, remembering that $\tilde{N}$ kernel modules, two dot product modules, one coherence module and one α update module are used. Table 5.1 shows the resource usage for each module and the total for the whole design. Since $\tilde{N}$ kernel modules are required, the kernel modules will dominate for large values of $\tilde{N}$ and M. The worst case scalability of the resource usage of the complete design is $\mathcal{O}(\tilde{N}M)$. The extra operations required to implement DKNLMS-CT are also shown in Table 5.1. Note that the number of correction terms are proportional to the delay factors, $d_{\mathcal{D}}-1$ and $d_{\alpha}-1$ respectively. The kernel vector evaluation scales the worst, $O(\tilde{N}M)$, while the

TABLE 5.2: Expected propagation delay of each module and the complete design

Module	exp	$\times$	$+$	$\div$	$<$
Kernel	1	2	$\lceil \log_2(M) \rceil + 1$	0	0
Dot Product	0	1	$\lceil \log_2(\tilde{N}) \rceil$	0	0
Coherence Criterion	0	0	0	0	1
$\boldsymbol{\alpha}$ Update	0	2	2	1	0
Design Total	1	5	$\lceil \log_2(\tilde{N}) \rceil + \lceil \log_2(M) \rceil + 4$	1	1
Correction Terms ($d_{\mathcal{D}}$)	1	2	$\lceil \log_2(M) \rceil + 1$	0	1
Correction Terms ($d_{\boldsymbol{\alpha}}$)	0	2	$\lceil \log_2(\tilde{N}) \rceil + \lceil \log_2(d_{\boldsymbol{\alpha}}) \rceil$	0	0

number of correction terms will also incur significant overhead if $d_{\mathcal{D}} - 1$ and $d_{\boldsymbol{\alpha}} - 1$ are large.

The expected propagation delay of each module and the whole design are given in Table 5.2. The propagation delay is expressed as a sum of arithmetic operations along the critical path. The overall expected propagation delay is the sum of the expected propagation delay of each arithmetic operation which lie on this critical path. Using this basic approximation of propagation delay, we can decide which optional registers to enable in the design, shown in Figures 5.4 to 5.8, or in the arithmetic operations themselves. Furthermore, $\boldsymbol{\alpha}$ update and coherence criterion modules appear to have constant latency, while the kernel and dot product modules scale as $\log_2(M)$ and $\log_2(\tilde{N})$ respectively. As such, the worse case scalability of the expected propagation delay is $\mathcal{O}(\log_2(M) + \log_2(\tilde{N}))$. In practice, larger designs will also likely suffer from placement and routing issues which will also affect the propagation delay. The expected propagation delay of the correction terms are also shown in Table 5.2. For the most part, the correction terms are computed in parallel with other modules, so the overall propagation delay does not change with the inclusion of correction terms. The exclusion for this, is the $\lceil \log_2(d_{\boldsymbol{\alpha}}) \rceil$ which increases the overall expected propagation delay.

5.5 Implementation

In this section, several details / methods related to the implementation of the different DKNLMS configurations are specified. Chisel v2.2.27 (Bachrach et al., 2012) was used to create a core generator capable of generating all the architectures described in Section 5.4. Chisel is a hardware construction language embedded in Scala which generated Verilog. It's a low abstraction domain specific language (DSL) which benefits from meta-programming in Scala to allow core generators to be made easily. Being a low abstraction DSL, Chisel allows complete control over the datapath while allowing the design to be scalable. As such, a design made in Chisel is comparable with an RTL design, but with increased productivity. All bitstreams and implementations in this work were generated using Vivado 2017.4 with aggressive optimisations enabled. Fixed point arithmetic was used throughout the design, unless specified otherwise, a wordlength of 18bits was used, although the design is parameterised for any fixed point format. The integer length was chosen to avoid overflows on the training dataset and the specific algorithm hyperparameters. In order to minimise the overhead of arithmetic operations, all fixed point arithmetic was implemented with a custom fixed point library in which the input and output of every arithmetic operation was the same fixed point format. When necessary, conversion from higher precision to lower precision formats, e.g., after a multiplication, was achieved using simple truncation. Furthermore, no saturation logic was implemented to avoid overflows. 18bit wordlengths were chosen as they map well to DSP blocks within FPGAs, while providing little degradation in the accuracy of the algorithm, this is explored further in Section 5.6. A further reduction in bitwidth may be achieved with a finer grained approach to bitwidth and integer length selection, but this is not explored in this work. The exponential evaluation and division operations can consume a significant amount of hardware, and cause a significant increase in the overall expected propagation delay. As such, signification attention was paid to the implementation of these operations.

5.5.1 Exponential and Division Approximation

The exponential and division operators represent areas where significant performance improvements can be made to the DKNLMS designs. Some specific improvements made to this design were based on the following observations:

- the range of the inputs is significantly less than the range of the fixed point datatype;
- both functions are differentiable;
- division can be implemented as an inversion and a multiplication; and
- the accuracy of DKNLMS is not very sensitive to approximations in both functions.

Given the above, several different implementations were considered, including:

1. a simple lookup table;
2. a lookup table with linear interpolation; and
3. the Remez algorithm (Tawfik, 2005) implemented using Estrin's polynomial evaluation method (Estrin, 1960).

The exponential function was implemented as a simple single input function approximation, while division was implemented as a reciprocal function, followed by a multiplication. Although the Remez algorithm and linear interpolation methods can provide many benefits, the rest of the design already used a significant amount of DSP resources and as such, a simple lookup table was used for both reciprocal and exponential functions with 1024 elements, we found this sufficient to provide good accuracy. We further discuss the implications on accuracy in Section 5.6.1.

5.5.2 Register Parameterisation

As the number of pipeline stages directly affects d, the more pipeline stages are added the more a DKNLMS variant will behave differently to KNLMS. As such, for small values of d the *placement* of the registers becomes vitally important. As stated in Section 5.4, the design was created with optional registers placed between each operation. When a particular core is generated, an array of Boolean values control whether or not a particular set of registers (i.e., a pipeline stage) is actually used in the design. This directly controls the delay parameters of DKNLMS and the variants, i.e., d, $d_{\mathcal{D}}$ and d_{α}. Also, when operations themselves are pipelined, shift registers are created to ensure that other signals (e.g., the valid signal) arrive at subsequent locations at the correct times.

In order to generate the array of Boolean values, we break the design up into approximate unit delays. Let us denote a unit delay with τ. For the given 18bit fixed point implementation, the estimated propagation delay of several operations is as follows:

- multiplication, τ;
- addition/subtraction, τ;
- exponentiation, 3τ; and
- division, 4τ.

The above delay values were approximated by synthesising and place & routing the individual arithmetic operators on the target FPGA device. The pipeline stages are then placed so they have approximately equal delay values separating them. For example, using Table 5.2, one can determine the expected propagation delay of each submodule. This estimate can then be used to decide which pipeline stage to enable to best improve the performance of the design. This is done by enabling the pipeline register which produces paths with the smallest possible estimated propagation delay.

5.6 Results

In this section, we compare the accuracy and performance of all the KNLMS-like algorithms discussed within: KNLMS, DKNLMS, MDKNLMS, DKNLMS-DG and DKNLMS-CT. For accuracy, we study the effect of delayed model adaptation on the modelling capability of the KNLMS. Issues with convergence, stability and model size when delayed model adaptation is introduced are also explored. In terms of performance, the throughput of the architectures described in Section 5.4 are provided for varying dictionary size ($\tilde{N}$), feature length (M) and latency (d). These performance figures are compared to a C implementation of KNLMS running on a PC.

5.6.1 Accuracy

In this subsection, the accuracy the DKNLMS algorithms are analysed empirically. In the first part of this section, we study the accuracy, convergence and stability of all DKNLMS variants. Firstly, we consider the DKNLMS variants using floating point arithmetic and without any of the function approximations described in Section 5.5. The extra effect of using fixed point arithmetic and function approximations is then considered in Section 5.6.1. For the accuracy experiments in this work, the chaotic Mackey-Glass (Mackey, Glass, et al., 1977) time series was used. The Mackey-Glass time series benchmark is generated using the following differential equation: $dx(t)/dt = -ax(t) + bx(t-\tau)/(1 + x(t-\tau)^{10})$ with $(a = 0.1, b = 0.2, \tau = 30)$, this configuration is identical to the benchmark used by Engel, Mannor, and Meir (2004). Our implementation was based off the KAF toolbox (KAFBOX) (Van Vaerenbergh, 2012).

Hyperparameter Optimisation

In practice, we suggest optimising the parameters for the DKNLMS variants independently of each other, allowing tradeoffs for convergence time and stability to be captured within the cross validation process. In order to adequately compare the different algorithms, hyperparameter optimisation must be performed for each algorithm. Otherwise, we may accidentally compare one algorithm at optimal settings to another at sub-optimal settings. The problem of hyperparameter optimisation is non-convex and therefore, we cannot be certain of achieving optimal settings, but a simple random search (Bergstra and Bengio, 2012b) will help us avoid extremely sub-optimal settings. We can also quantify our effort in finding good hyperparameters in terms of the number of samples required to achieve good accuracy. In this work, we use the following hyperparameter optimisation procedure:

1. 10-fold cross validation was used, separating the training set into ten different training and validation sets;
2. for each training example, the resultant model was tested, using MSE, on the entire validation set to produce a model convergence series;
3. after removing an initial number of examples during the convergence period, the remaining entries in the model convergence series are averaged; and
4. the model with the lowest average MSE in the convergence period, was assumed to have the best hyperparameters.

For the examples in Section 5.6.1 the training set size was 1000 and the convergence period was 500 samples. When there was negligible accuracy difference, sometimes the 2nd best hyperparameters where used if they provided: 1) a more compact model; or 2) a more stable convergence series. Table 5.3, shows the optimised hyperparameters for each DKNLMS configuration, along with average model size, $\tilde{N}$, and average MSE over the cross validation procedure. The baseline average MSE in the final columns uses some common values for each of the hyperparameters of KNLMS without specific optimisation for each algorithm. Specifically, the values of the hyperparameters for the baseline were as follows: $\gamma = 1.39$, $\eta = 0.1$, $\epsilon = 10^{-4}$ and $\mu_0 = 0.9$. Interestingly, when the baseline results are compared, we get very similar performance between KNLMS, DKNLMS ($d \leq 8$) and MDKNLMS ($d \leq 16$), while DKNLMS-DG suffers significant accuracy degradation for all values of d. As expected, all algorithms for all values of d perform better when their hyperparameters are optimised specifically for them. By comparing the baseline and optimised results for DKNLMS-DG it is clear this algorithm behaves very differently to the original KNLMS algorithm. Through hyperparameter optimisation, the accuracy of the DKNLMS-DG variants is significantly improved to the point where it appears to perform better than the other variants for $d = 4$ and $d = 8$. However, it should be noted that since the behaviour of DKNLMS-DG was significantly different to the others, more time was spent optimising the hyperparameters in order to find acceptable values. The search space was broader and roughly $20\times$ points were sampled, compared to the other configurations. If the other algorithms were allowed a larger search space, this difference may be less pronounced. Next, the baseline DKNLMS ($d = 32$) appears to have unstable behaviour, but, after hyperparameter optimisation it appears to have achieved stability with some degradation in accuracy compared KNLMS. As expected, DKNLMS-CT performs almost identically to KNLMS.

TABLE 5.3: Cross-validation accuracy and hyperparameters for each DKNLMS variant

Algorithm	d	γ	η	ϵ	μ_0	Optimised Average $\tilde{N}$	Optimised Average MSE ($\times 10^{-2}$)	Baseline Average $\tilde{N}$	Baseline Average MSE ($\times 10^{-2}$)
KNLMS	-	1.48	0.15	0.050	0.69	19.9	1.37	87.3	1.49
DKNLMS	4	1.48	0.15	0.050	0.69	25.7	1.35	88.2	1.44
	8	1.48	0.15	0.050	0.69	26.1	1.31	88.7	1.37
	16	0.88	0.068	0.090	0.75	23.6	1.58	88.3	1.61
	32	2.14	0.036	0.061	0.77	66.9	1.60	103.4	3.16×10^5
MDKNLMS	4	1.23	0.13	0.063	0.87	54.1	1.53	88.2	1.54
	8	1.48	0.15	0.050	0.69	25.7	1.41	88.2	1.44
	16	1.48	0.15	0.050	0.69	26.1	1.32	88.7	1.39
	32	0.89	0.063	0.089	0.85	32.9	1.63	88.3	1.70
DKNLMS-DG	4	1.36	0.27	2.55	0.94	128	1.28	90.5	2.07
	8	2.21	0.38	3.06	0.70	36.7	1.21	80.3	4.23
	16	1.92	0.49	3.72	0.42	9.1	1.49	66.7	6.96
	32	3.24	0.72	3.89	0.28	10	2.28	48.3	4.60
DKNLMS-CT	4	1.48	0.15	0.05	0.69	19.9	1.37	87.3	1.49
	8	1.48	0.15	0.05	0.69	19.9	1.37	87.3	1.49
	16	1.48	0.15	0.05	0.69	19.9	1.38	87.3	1.50
	32	1.48	0.15	0.05	0.69	19.9	1.40	86.9	1.53

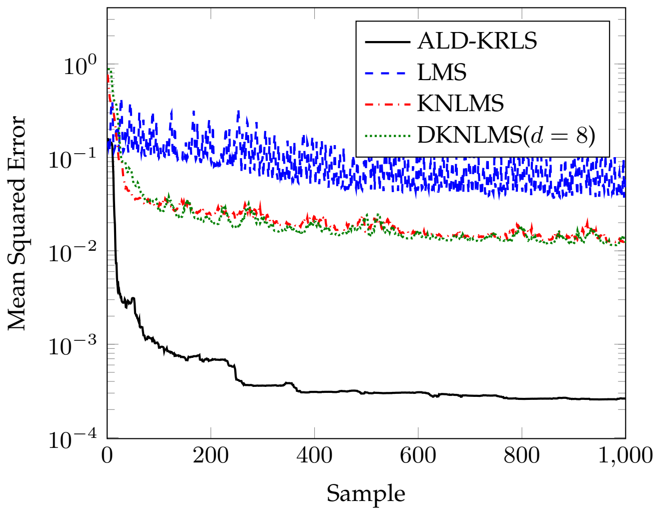


FIGURE 5.9: Comparison of prediction performance between DKNLMS ($d = 8$) and other adaptive filtering algorithms.

Accuracy Comparison

The cross validation accuracy is useful for hyperparameter optimisation, but the real test is the accuracy on the test set. Furthermore, when dealing with time series data, it may be necessary to plot the convergence of a learning algorithm to better understand its behaviour. In this subsection, we plot the convergence of each algorithm, firstly with the baseline hyperparameters, secondly, with optimised hyperparameters and compare the differences.

However, before we look closely at the proposed algorithms, let us first put some of the results in context. Figure 5.9 shows the accuracy of DKNLMS ($d = 8$) versus KNLMS and some other common adaptive filtering algorithms in the literature. Firstly, we can see that ALD-KRLS (Engel, Mannor, and Meir, 2004) outperforms

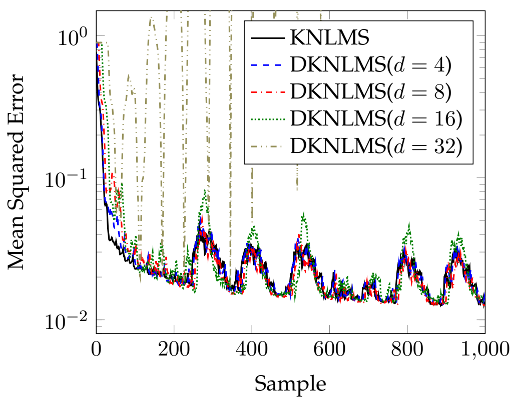


FIGURE 5.10: KNLMS vs DKNLMS using the same parameters for each configuration.

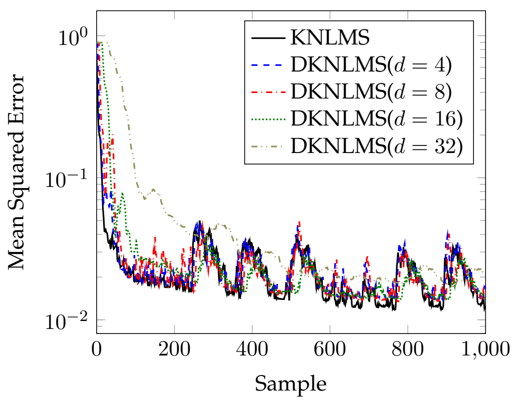


FIGURE 5.11: KNLMS vs DKNLMS with parameters adjusted for each configuration.

KNLMS by almost two orders of magnitude. However, the increased complexity of the dependencies of ALD-KRLS mean that it's not the subject of our study for now. The purpose of the figure is to provide context around different types of adaptive filters and their costs and benefits. ALD-KRLS scales with $O(\tilde{N}^2 + \tilde{N}M)$ in operations and memory, as such it would be a much more difficult task to implement at sample rates in the order of hundreds of MHz as we do in this work. Secondly, we see that KNLMS outperforms LMS (Widrow and Hoff, 1960) by almost an order of magnitude. Currently, if high frequency (in the order of 100s of MHz) adaptive filtering is required, variants of LMS are one of the few available options.

Figures 5.10 and 5.11 show the convergence pattern of DKNLMS while varying the delay parameters. Figure 5.10 uses the baseline hyperparameters, while Figure 5.11 uses the optimised hyperparameters for each configuration. Figure 5.10 shows that for DKNLMS ($d \leq 8$), the accuracy and convergence are very similar to KNLMS. For $d = 16$, some instability begins to show, while for $d = 32$ DKNLMS becomes completely unstable. In Figure 5.11 we see the stability for $d = 32$ significantly improve, at the cost of convergence speed. For $d = 16$ with optimised hyperparameters, $d = 16$ seems to perform effectively as good as KNLMS for this benchmark.

Moving on to the second variant, MDKNLMS, Figures 5.12 and 5.13 show the convergence pattern of MDKNLMS while varying the delay parameters. Again, Figure 5.12 uses the baseline hyperparameters, while Figure 5.13 uses the optimised hyperparameters for each configuration. Figure 5.12 shows similar convergence for all configurations of MDKNLMS ($d \leq 16$), while some instability begins to show when $d = 32$. After hyperparameter optimisation, all configurations of MDKNLMS perform well, with $d = 32$ only show a slight deterioration of convergence speed.

Now for DKNLMS-DG, Figures 5.14 and 5.15 show the convergence pattern of DKNLMS-DG while varying the delay parameters. Similar to the previous examples, Figure 5.14 uses the baseline hyperparameters, while Figure 5.15 uses the optimised hyperparameters for each configuration. In Figure 5.14, we see a significant difference between the accuracy of KNLMS and DKNLMS-DG for all factors of d. In terms of accuracy, all configurations are visibly less accurate and have slower convergence than KNLMS. For $d \geq 16$, the DKNLMS-DG also show visible instability in the convergence region, without exhibiting the extremely unstable behaviour of

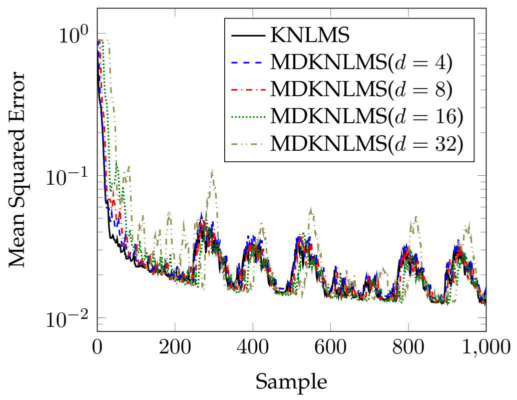


FIGURE 5.12: KNLMS vs MDKNLMS using the same parameters for each configuration.

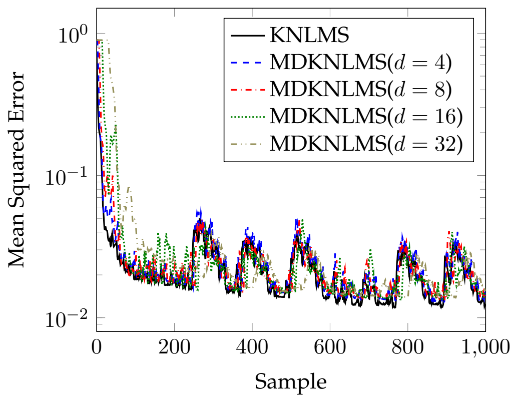


FIGURE 5.13: KNLMS vs MDKNLMS with parameters adjusted for each configuration.

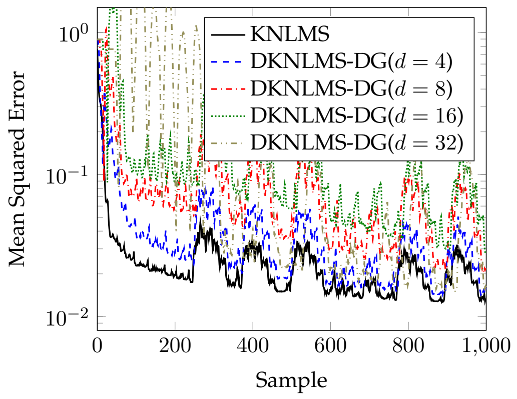


FIGURE 5.14: KNLMS vs DKNLMS-DG using the same parameters for each configuration.

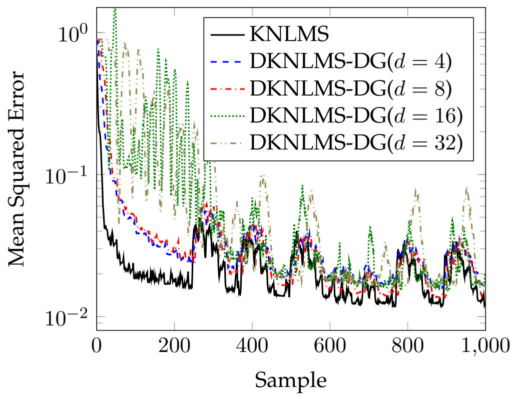


FIGURE 5.15: KNLMS vs DKNLMS-DG with parameters adjusted for each configuration.

DKNLMS ($d = 32$). After hyperparameter optimisation, shown in Figure 5.15, all configurations of DKNLMS-DG show significantly better accuracy, at the expense of significantly worse convergence speeds. Furthermore, even after hyperparameter optimisation, instability can be seen when $d \geq 16$. We suspect this instability may be caused by the fractional part of Equation (5.9) tending towards zero, which can occur for DKNLMS-DG configurations. We also note that the ϵ value that was found for DKNLMS-DG configurations was significantly higher than the other DKNLMS variants, which may be to counteract this effect. Overall, the accuracy results of the DKNLMS-DG configurations were underwhelming, and as such will often not be used for comparisons in the subsequent sections.

Finally, Figures 5.16 and 5.17 show the convergence pattern of DKNLMS-CT while varying the delay parameters. Again, similar to previous examples, Figure 5.16 uses the baseline hyperparameters, while Figure 5.17 uses the optimised hyperparameters for each configuration. As expected, DKNLMS-CT shows the exact same convergence pattern as KNLMS itself, just shifted by the delay amount d. While from an accuracy

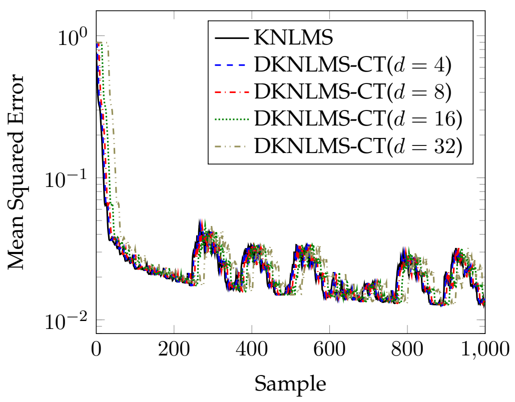


FIGURE 5.16: KNLMS vs DKNLMS-CT using the same parameters for each configuration.

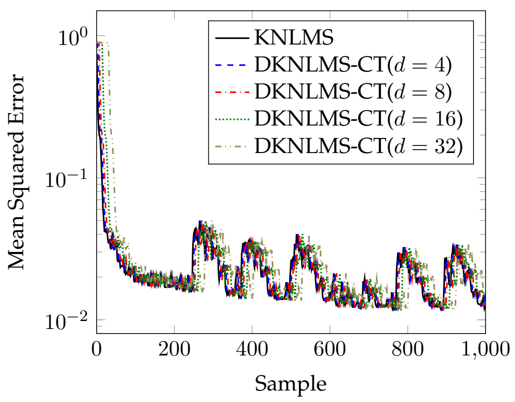


FIGURE 5.17: KNLMS vs DKNLMS-CT with parameters adjusted for each configuration.

point of view, DKNLMS-CT is promising, one must remember that this comes at the cost of increased hardware resources and increased critical path delay. In practical terms, this means reduced dictionary size and higher latency. Consider the results shown in Table 5.3 for MDKNLMS and DKNLMS-CT when $d = 16$. Rounding the values, MDKNLMS / DKNLMS-CT would require dictionary sizes of 20 / 26 respectively. Using the feature length of Mackey-Glass of $M = 7$, and the arithmetic counts in Table 5.1, the number of operations per update can be calculated for MDKNLMS / DKNLMS-CT as 756 / 1023 respectively, assuming $d_{\mathcal{D}} = d_{\alpha} = 8$. In all, that would mean that for this example DKNLMS-CT would require a 35% increase in Ops for no accuracy benefit over MDKNLMS. Although this may be useful in some circumstances, in this benchmark we do not see a significant advantage in using DKNLMS-CT over DKNLMS or MDKNLMS.

Now that each DKNLMS variant has been compared against KNLMS, let us now compare these variants against each other. Figures 5.18 and 5.19 shows DKNLMS and MDKNLMS convergence patterns plotted directly against each other using high values of d. Specifically, Figure 5.18 uses $d = 16$ while Figure 5.19 uses $d = 32$. In Figure 5.18 we see both algorithms working quite well at $d = 16$, with MDKNLMS perhaps showing a small amount of instability. In Figure 5.19 we see a larger difference between each configuration, both DKNLMS variants exhibiting slower convergence speeds than KNLMS, DKNLMS being much slower than MDKNLMS. More importantly, DKNLMS clearly shows some accuracy degradation when $d = 32$, while MDKNLMS behaves very similar to DKNLMS. Given this, we consider MDKNLMS to be the most promising DKNLMS variant when it comes to achieving high throughput learning models. In the subsequent sections, in order to save space we'll mostly provide results for MDKNLMS.

Finite Precision and Function Approximation

In this section, we test accuracy with a full architecture simulation, as such finite precision effects and function approximation is taken into account. For the Mackey-Glass time series and the hyperparameters described in Section 5.6.1 a wordlength of 18bits with a 5bit integer length was used. Figure 5.20 shows the modelling accuracy of the floating point vs several different fixed point versions with function approximations.

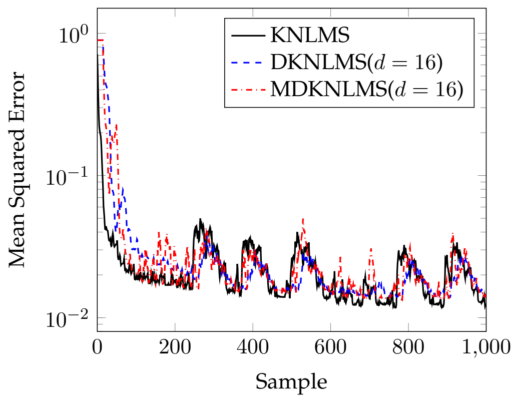


FIGURE 5.18: DKNLMS and MDKNLMS when $d = 16$.

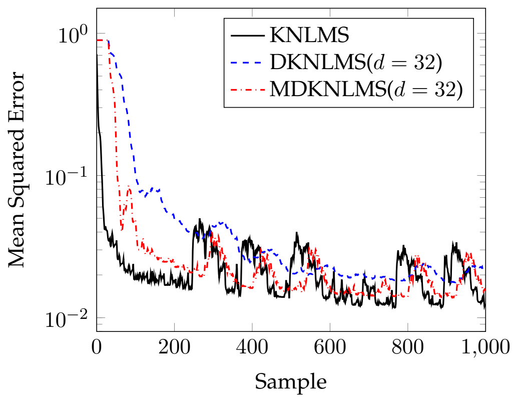


FIGURE 5.19: DKNLMS and MDKNLMS when $d = 32$.

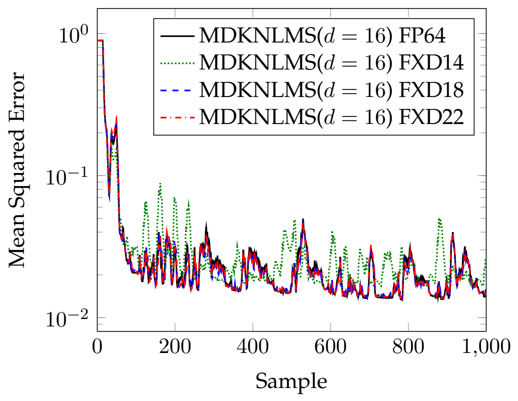


FIGURE 5.20: Comparison between double precision floating point (FP64) and fixed point (FXD) MDKNLMS when $d = 16$.

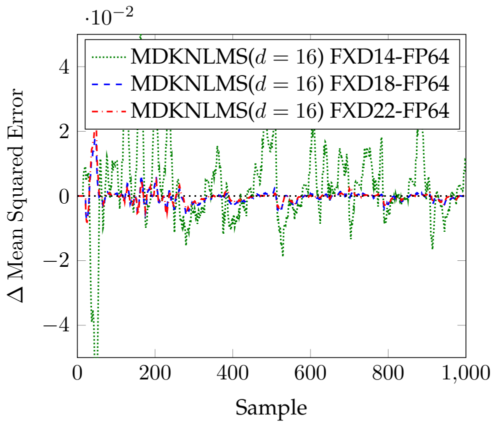


FIGURE 5.21: Accuracy difference between double precision floating point and fixed point (FXD) MDKNLMS when $d = 16$.

Note, all fixed point formats use 5bits for the integer length which avoids overflow and Mackey-Glass time series using MDKNLMS ($d = 16$). Figure 5.21 shows the difference in accuracy between the floating point and fixed point versions. Note, floating point refers to double precision, i.e., 64bit float. In Figure 5.20 we can see that using 14bit fixed point arithmetic causes some accuracy issues, affecting the stability of MDKNLMS. Meanwhile, 18bit and 22bit does not significantly affect the accuracy of MDKNLMS ($d = 16$). In terms of quantifying the change in modelling error, Figure 5.21 shows the accuracy difference between fixed point and floating point formats. In particular, with reference to both Figures 5.20 and 5.21, we can see that the difference in modelling accuracy is approximately an order of magnitude lower than the modelling error incurred by the algorithm itself, for both 18bit and 22bit fixed point formats. The 14bit format introduces some error which would noticeably affect the overall accuracy of MDKNLMS. Interestingly, in many parts of the convergence curve, the 18bit and 22bit fixed point models actually outperform the floating point model, in terms of accuracy. This could be due to the rounding error perhaps

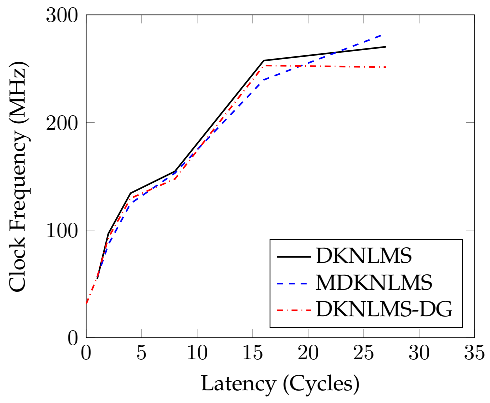


FIGURE 5.22: Throughput versus latency (clock cycles) for $\tilde{N} = 64$ and $M = 8$ for all DKNLMS variants

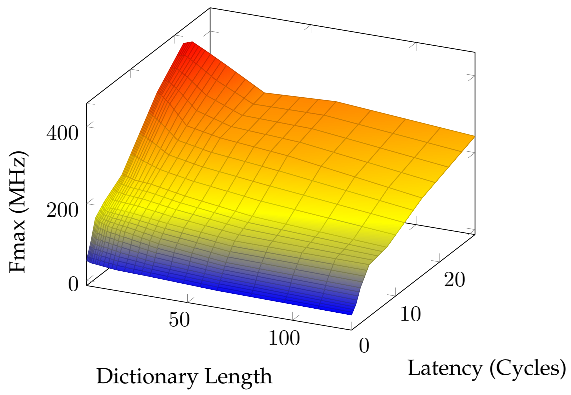


FIGURE 5.23: Throughput versus latency (clock cycles) for $M = 8$ for MDKNLMS

behaving as a regulariser in this model, by injecting noise in the form of quantisation error. This injected quantisation could actually behave like $l2$ regularisation and help the algorithm avoid overfitting (Bishop, 1995). Similar effects have been observed by prior works, relating quantisation error to noise injection (Baskin et al., 2018b; Baskin et al., 2018a) and relating quantisation to regularisation (Courbariaux, Bengio, and David, 2015). Further study of this effect is future work.

5.6.2 Performance

In this section, we look at the potential performance and scalability of the DKNLMS architectures described Section 5.4. This section contains post-place-and-route results on a Xilinx VU9P FPGA. Full system implementations are found in Section 5.6.4.

Figure 5.22 shows how the throughput of DKNLMS, MDKNLMS and DKNLMS-DG varies with the latency (clock cycles) of the design. For $\tilde{N} = 64$ and $M = 8$, $d = 4$ pipeline stages are required to achieve a throughput of 100 MHz and a throughput of 280 MHz can be achieved if all of the optional registers described in Section 5.4 are enabled, i.e., $d = 31$. These represent speedups of 4.4×/12.0× over a combinatorial KNLMS design (i.e. $d = 1$), respectively. We do see a slight difference in maximum clock frequency between the different DKNLMS variants, but this appears to be some noise from the place-and-route tool, rather than any particular trend. Furthermore, Figure 5.23 provides insights into the space of frequencies which can be achieved for different values of $\tilde{N}$ and d. Overall, frequencies up to 420 MHz can be achieved for high latencies ($d \geq 26$) and low dictionary sizes ($\tilde{N} \leq 4$). However, with reference to Section 5.6.1, for more practical values of $\tilde{N}$ ($32 \leq \tilde{N} \leq 64$), and d ($16 \leq d \leq 32$), frequencies of between 214 MHz to 296 MHz can be achieved.

5.6.3 Area Usage

Figures 5.24 and 5.25 show the DSP usage as a function of M and $\tilde{N}$. All of the DKNLMS variants have the same DSP usage so this chart is representative of the entire design space described in Section 5.4. Figures 5.24 and 5.25 show that the DSP usage is entirely predictable across the different DKNLMS variants. The DSP usage corresponds *exactly* to the amount of multiplies specified in Table 5.1 until the number

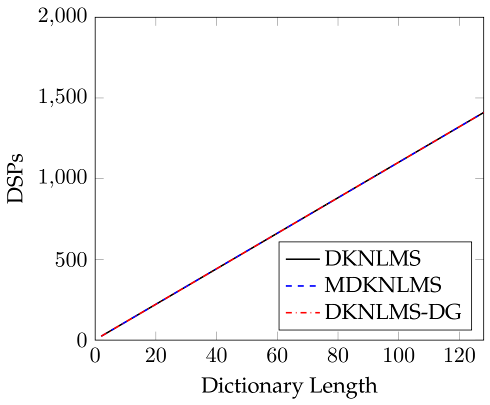


FIGURE 5.24: DSP usage when $M = 8$ for all DKNLMS variants

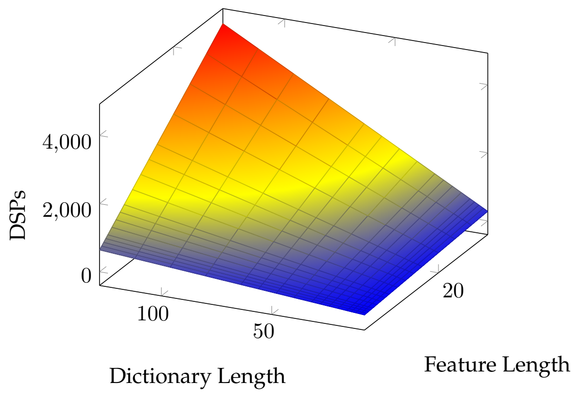


FIGURE 5.25: DSP usage while scaling $\tilde{N}$ and M for MDKNLMS

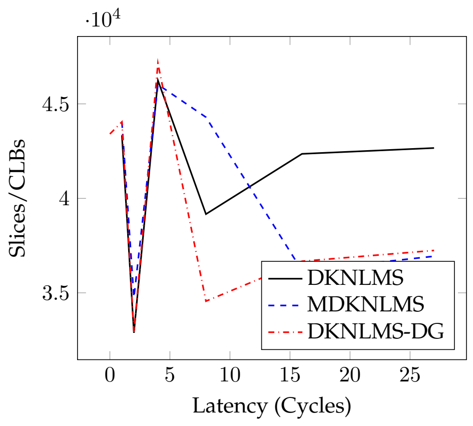


FIGURE 5.26: Resource usage versus latency (clock cycles) for $\tilde{N} = 64$ and $M = 8$ for all DKNLMS variants

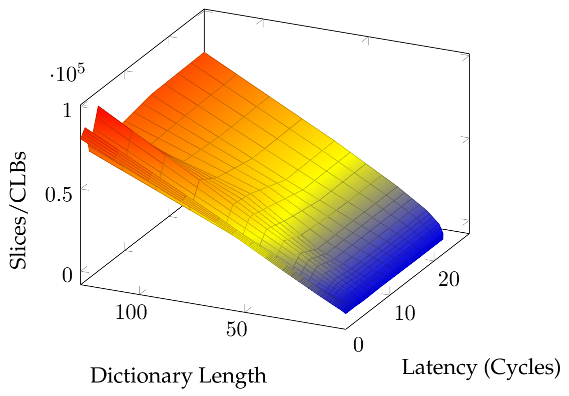


FIGURE 5.27: Resource usage versus latency (clock cycles) for $M = 8$ for MDKNLMS

of DSPs increases beyond what is available in the target device, in which case the extra multipliers are implemented in LUTs.

Figures 5.26 and 5.27 show the LUT usage as a function of M and $\tilde{N}$. Note that in this figure the resource of DKNLMS with all optional registers enabled is shown. The other designs very closely match the resource usage shown in this figure, within 20%. Interestingly, the resource usage is higher when $d = 4$ than when $d = 8$ we suspect this is due to the routing tool replicating parts of the design to try to shorten the critical path. Clearly, the LUT usage is significantly less than the DSP usage (when one considers the availability of such resources on an FPGA device), meaning that we'll be DSP bound as we scale.

5.6.4 Implementation and System Performance

In this section, we consider post-synthesis results along with system level performance, latency and performance measurements. For comparison, we use two CPU-based platforms: (1) a laptop PC running Ubuntu Linux 14.04 with an Intel Core i7-4500 CPU running at 1.8GHz and GCC 4.8; and (2) an Ultra96 board from Agilent, which features an ARM Cortex-A53 CPU running at 1.2GHz and GCC 6.2. For the CPU implementations, a C library was created to implement KNLMS. During testing, ATLAS (Whaley and Petitet, 2005), OpenBLAS (Wang et al., 2013) (both with and without multi-threading) and a single-threaded hand-coded library were used to provide the linear algebra routines. For each test, the fastest implementation is reported, which was always the hand-coded library. Although the hand-coded library was written in pure C, care was taken to ensure that GCC's auto-vectorisation capabilities could effectively optimise the code. This includes implementing loops with compile-time static loop bounds on many loops and avoiding irregular memory access patterns. We examined the assembly code of several routines (including the dot product) and ensured they were significantly unrolled and made use of the processor's SIMD instructions. Our C implementation was compiled using -O3, -ffast-math and several other optimisation flags to improve performance. Surprisingly, using highly tuned linear algebra libraries and multi-threading did not provide any performance benefit, we suspect the small vector sizes used throughout meant the overheads of multi-threading outweighed the benefits. Furthermore, the fine grained parallelism available in the KNLMS algorithm make it difficult to parallelise on CPUs, for a single model. Although we were unable to extract any extra performance by going to multiple cores/threads, there is still a theoretical benefit, as such all CPU results in this section could possibly be improved by a factor c, the number of cores available in the CPU, particularly if training of multiple parallel models is required. Also, CPU implementations of DKNLMS (and the other variants) in C did not produce higher performance, therefore we only provide the performance results of KNLMS when running on the CPU. In terms of fabrication process, the i7-4500 CPU is 22nm, while the FPGA devices, an XC7Z020 and an XCZU3EG, are 28nm and 16nm respectively.

The FPGA implementations in this section fall into two categories:

(1) post-synthesis estimates

(2) system implementations

The post synthesis estimates are post-place-and-route results of the MDKNLMS cores only. The system implementations utilise AXI master to stream, AXI stream to master and generated drivers from fpga-tidbits[2] to move data between the DDR memory of the ARM host and the accelerator core and to manage the core. The system implementation also contains an extra FIFO of length d to allow the stream to AXI controller to apply backpressure to the core. For the system implementation, the time series data starts and finishes on the host and this transfer time is included in the timing measurements for system performance. Table 5.4 shows our results for MDKNLMS ($d = 16$), along with key results of several previous works. In particular, note that throughput refers the sample rate for a *single model* for which the implementation is able to operate at. For our designs, this sample rate also corresponds to the clock rate of the FPGA design. As with many machine learning algorithm families, KAFs come in various different forms and target different application domains. As such, readers should be wary when directly comparing results as the specific targets of each

[2]https://github.com/maltanar/fpga-tidbits

work may not be reflected in Table 5.4. Given this, where possible the results from previous works are shown which closely match the algorithm and parameters used in this work. The throughput / latency numbers for the post-synthesis results refer to the sample rate and pipeline depth of the core. For the system implementations, the throughput / latency numbers refer to measured numbers from the host code (C++ code running on the ARM core). Finally, the performance, measured in GOps/s, is calculated as the number of Ops / Update for each new sample of the adaptive filter, multiplied by the number of samples per second that can be processed by the hardware, i.e., the throughput.

For power measurements, for the Ultra96 platform board power is measured using device reported board power measurements.[3] KNLMS and MDKNLMS were run in a loop for approximately 10 minutes, with power measured every 10 seconds. The reported power is the average of the power measurements over this period. For the Pynq-Z1 platform, board power is measured using an inline USB power meter. Again, average power is reported over a 10 minute test. For the PC platform, energy dissipation was measured using OS reported battery charge over a 10 minute test with the display switched off and the CPU governor set to "performance", to prevent throttling of the CPU clock.

Looking at the system implementations in Table 5.4, The CPU implementations achieve 0.66 / 0.084 GOps/s for the Intel i7 / ARM Cortex-A53 respectively. The architectures proposed by Ren et al. (2014) and Fraser et al. (2017a) were designed for multi-channel systems, or hyperparameter optimisation, and therefore perform poorly when constrained to accelerating a single channel. For Fraser et al. (2017a), this translates into 0.50 GOps/s of floating-point performance. Although the hardware by Pang et al. (2016) is designed to accelerate single-channel models, the authors target flexibility over performance. This means that the most comparable design ($\tilde{N} = 64$, $M = 7$) achieves relatively low utilisation: a floating-point performance of 0.52 GOps/s. It should be noted, that the design proposed by Pang et al. (2016) can be scaled to larger problems than this work. Furthermore, the design achieves better performance for RLS-style kernel methods, such as FB-KRLS (Van Vaerenbergh et al., 2010) and SW-KRLS (Van Vaerenbergh, Via, and Santamaria, 2006).

When comparing similar sized problem sizes, starting with $\tilde{N} = 30$ and $M = 7$, the Pynq-Z1 implementation achieves throughput 702 / 83× higher than the ARM / i7 implementations respectively. Similarly, it outperforms the implementation by Pang et al. (2016), achieving 165× higher peak performance in GOps/s. When power is taken into consideration, the Pynq-Z1 design consumes the least board power of all designs 1.27 / 6.55× less than the ARM / i7 implementations respectively. Finally, the energy consumed per update for the Pynq-Z1 design is 899 / 550× less than the ARM / i7 implementations respectively.

Moving on to the larger problem size, $\tilde{N} = 46$ and $M = 7$, the Ultra96 implementation achieves throughput 2975 / 360× higher than the equivalent ARM / i7 implementations respectively. For power, the ARM achieves 2.04× lower power consumption than the Ultra96 platform. Conversely, the i7 consumes 2.52× more power. In terms of energy, the Ultra96 is the most efficient design of all, achieving 1446 / 902× better efficient than the equivalent ARM / i7 implementations.

For architectures which are more similar, the design proposed in Fox et al. (2016) achieves very high throughput, 104.2 MHz. In comparison, our Ultra96 implementation achieves 187.4 MHz for a larger model, at 250 GOps/s at a slightly lower precision, Fixed24 $\rightarrow$ Fixed18. This corresponds to an increased data rate of 1.80×.

[3]Board power measured in μW by polling `/sys/class/hwmon/hwmon0/power1_inputcat`

TABLE 5.4: Comparison Between Different System Implementations of Various Kernel Adaptive Filters

Reference	Algorithm	Device	Dictionary Size	Feature Length	Datatype	Ops / Update	Latency (μs)	Sample Rate (MHz)	Board Power (W)	Performance (GOps/s)	Energy (μJ/Update)
Post-synthesis Estimates											
Tridgell et al. (2015)	NORMA	XC7VX485T	32	8	Fixed18	-	0.080	137.8	-	-	-
Tridgell et al. (2015)	NORMA	XC7VX485T	64	8	Fixed18	-	0.087	137.4	-	-	-
Fox, Boland, and Leong (2018)	Fastfood	XCKU035	16,384	1,024	Fixed18	-	4.4	0.42	-	-	-
Ours	MDKNLMS($d = 16$)	XC7Z020	30	7	Fixed18	872	0.26	66.7	-	58.16	-
Ours	MDKNLMS($d = 16$)	XCZU3EG	46	7	Fixed18	1336	0.10	187.5	-	250.5	-
System Implementations											
-	KNLMS	ARM Cortex-A53	30	7	Float32	872	10.5	0.095	4.1	0.083	43.16
-	KNLMS	ARM Cortex-A53	46	7	Float32	1336	15.8	0.063	4.1	0.084	65.08
-	KNLMS	Intel i7-4500U	30	7	Float32	872	1.3	0.80	21.1	0.694	26.38
-	KNLMS	Intel i7-4500U	46	7	Float32	1336	2.0	0.52	21.1	0.700	40.58
Ren et al. (2014)	KLMS	XC7V2000T	$53^\dagger$	1	Float32	-	-	2.4	-	-	-
Pang et al. (2016)	KNLMS	DE5	32	7	Float32	930	2.6	0.38	-	0.353	-
Pang et al. (2016)	KNLMS	DE5	64	7	Float32	1858	3.6	0.28	$22.32^\ddagger$	0.520	79.71
Fox et al. (2016)	NORMA	ExaNIC X4	14	8	Fixed24	-	-	104.2	-	-	-
Fraser et al. (2017a)	KNLMS	VC707	16	8	Float32	514	-	0.98	-	0.504	-
Ours	MDKNLMS($d = 16$)	Pynq-Z1	30	7	Fixed18	872	3.4	66.7	3.22	58.16	0.048
Ours	MDKNLMS($d = 16$)	Ultra96	46	7	Fixed18	1336	1.5	187.4	8.38	250.4	0.045

† the dictionary size is estimated by reimplementing the work
‡ power measurement taken for the SW-KRLS algorithm

TABLE 5.5: Resource Usage of System Implementations

	Algorithm	Device	Dict. Size	Feat. Len.	Datatype	DSPs (%)	LUTs (%)	BRAM (18k) (%)	Fmax (MHz)
	MDKNLMS($d = 16$)	Pynq-Z1	30	7	Fixed18	220 (100%)	42473 (79.84%)	16 (5.71%)	66.7
Dev. Tot.	-	XC7Z020	-	-	Fixed18	220	53200	280	-
	MDKNLMS($d = 16$)	Ultra96	46	7	Fixed18	360 (100%)	54437 (77.15%)	37 (8.56%)	187.5
Dev. Tot.	-	XCZU3EG	-	-	Fixed18	360	70560	432	-

The Pynq-Z1 also achieves respectable performance at 58 GOps/s. Note, we consider all operations in Table 5.1, so exponential and division counts as a single operation, even though they are lightweight table lookups in practice. Finally, the measured latency of our designs are significantly higher than the theoretical latency of the cores themselves. This is due to a combination of DRAM latency and batching that occurs in our system implementation, These overheads would not occur in implementations which are directly coupled to sensors.

Resource Usage

The resource usage of our system implementations is given in Table 5.5. These numbers reflect the system designs reported in Table 5.4. Our designs utilise all of the available DSPs on each device, after which multipliers are mapped to LUTs and quickly fill the available LUT resources. For both designs, we scale the core to fill almost all of the available device resources. Since the design is latency (specifically, clock cycle latency) sensitive, the overall Fmax is not affected too much by scaling the design. While the performance numbers are respectable for both the Pynq-Z1 and Ultra96 boards, the Fmax for all designs is quite low. Even High-Level Synthesis (HLS)-based designs can usually achieve 100 / 300 MHz on the XC7Z020 / XCZU3EG respectively. This again, is due to the sensitivity of the design to latency. Each design is a core containing 16 pipeline stages, which utilises over 75% of the available LUTs on each device. For comparison, Table 5.5 also contains the total resources available on each device for the Pynq-Z1, Ultra96 platforms respectively.

5.7 Conclusion

In conclusion, in this work a new technique (delayed model adaptation) has been proposed for modifying KAFs. The technique significantly reduces the dependency problem of KAFs (due to their recursive nature) allowing for high throughput hardware implementations. The delayed model adaptation technique is demonstrated by modifying the KNLMS algorithm. In doing so, several new variants of KNLMS are proposed which do not suffer from the same dependency problem. A core generator is described which is able to generate all variants of KNLMS. For similarly sized problems, the DKNLMS algorithms are able to achieve speedups of 360$\times$ over a CPU and peak performance 165$\times$ higher than a previous FPGA implementation. Our most performant design can operate at a data rate of 187.4 MHz and achieves a peak performance of 250 GOps/s. Furthermore, the design also achieves a 1.80$\times$ speedup over a prior FPGA implementation of NORMA. This work demonstrates that high throughput implementations of KAFs are achievable on current FPGA hardware, and hope this enables the use of KAFs in many more applications, with tight throughput requirements.

Finally, in comparison with other FPGA-based KAF designs, this work shows that one-size-fits-all solutions might not be an effective way of addressing the computational demands of machine learning algorithms. This is because machine learning algorithms are used in a vast number of application domains, with widely varying problem sizes and constraints. In particular, we show how very few prior works have addressed the high performant, single-channel online models as would be required for certain applications, such as channel equalisation. Furthermore, if those requirements are addressed specifically, and algorithmic modifications are made, significant gains can be attained.

Chapter 6

Distributed Kernel Recursive Least Squares

6.1 Introduction

The amount of data available continues to increase at an exponential rate (Szalay and Gray, 2006) and fast implementations of data mining primitives are essential to make sense of them. Parallelism is a common technique used to scale algorithms to Big Data problems. Unfortunately, many standard data mining techniques have dependencies which prevent them from being easily parallelised on a distributed platform. As a result, standard machine learning algorithms such as SVM and kernel based least-squares optimisers are often not considered suitable for large data mining problems.

One method for performing a parallel computation whilst minimising communication overhead is to split the data up into several subsets, each of which is used to create an independent *submodel*. If the learning algorithm is appropriate, the individual submodels may be good representations of their respective subsets, but they may not necessarily be a good representation of the entire data set. The core issue facing this technique then becomes how to combine the submodels into a single model.

An alternative method is to use a distributed platform to construct a single model. This approach differs from the submodel approach in that there is often a single model that is accessible from all computing nodes. The nodes then collectively optimise the model. A notable example is the paper by Chang (2011). The authors report a large speedup over LIBSVM (Chang and Lin, 2011). The main drawback of this approach is the communication overhead, which accounts for over 50% of the running time on some datasets for a large number of machines.

In this work, we take the approach of creating submodels and then combining them. We show that a closed form solution for combining individual Kernel Recursive Least Squares (KRLS) (Engel, Mannor, and Meir, 2004) submodels can be achieved. This result can then be simplified to a more compact form by applying the KRLS algorithm an additional time. We show that the final model is close to a serial, single module solution and only requires one-pass through the data. A theoretical analysis of the combined model is provided, along with an empirical analysis of its performance and accuracy in relation to standard benchmarks.

The main contributions of this work can be summarised as follows:

- a technique to combine multiple kernel based regression models on distributed nodes without requiring multiple passes over the data, furthermore, a large portion of the compute of this technique is *embarrassingly parallel*, meaning it can parallelised on loosely coupled accelerators;

- a theoretical bound on the approximation error in the representation of the kernel matrix while using this technique; and
- empirical analysis of its performance on benchmark datasets when compared to other batch and distributed techniques.

This chapter is organised as follows: Section 6.2 covers relevant background of the work. Section 6.3 describes our proposed algorithm including its theoretical formulation and implementation. Section 6.4 covers testing the methods on several different benchmark data sets. Section 6.5 provides and extension to the ALD criterion and a generalisation to multi-layered tree structures. Finally, conclusions are drawn in Section 6.6.

6.2 Kernel Recursive Least Squares

In this section we briefly survey the KRLS algorithm (Engel, Mannor, and Meir, 2004). Note, before describing KRLS, we briefly recap some kernel methods background from Section 2.1.

6.2.1 Kernel-based non-linear prediction

Consider a set of N observations in the form of input/output pairs $\{\mathbf{x}_n, y_n\},\ n \in [1, N]$, where the input entries $\mathbf{x}_n$ are vectors of length M. We refer to this data set as the *training data*. In a typical time series prediction scenario, the vector $\mathbf{x}_n$ consists of the few data samples that directly precede the value y_n. Given a new input entry, $\mathbf{x}$, kernel-based methods predict the corresponding output as:

$$\hat{y} = \sum_{i=1}^{N} k(\mathbf{x}_i, \mathbf{x})\, \alpha_i \ , \tag{6.1}$$

where $k(\cdot, \cdot)$ is a positive-definite kernel function.

Kernel-based prediction models thus consist of two types of data: a set of training input vector examples, which we refer to as the *dictionary*, and the corresponding coefficients α. The coefficients are typically calculated such that the prediction is as accurate as possible for the entire training data, in the least-square sense. In other words, the vector of the coefficients $\boldsymbol{\alpha} = [\alpha_1,\, \alpha_2,\, \ldots,\, \alpha_N]^T$ is defined as:

$$\boldsymbol{\alpha} = \underset{\boldsymbol{\gamma}}{\text{argmin}}\, \|\mathbf{y} - \mathbf{K}\boldsymbol{\gamma}\|^2 \ , \tag{6.2}$$

where $\mathbf{y}$ is the vector of the training output entries, $\mathbf{y} = [y_1,\, y_2,\, \ldots, y_N]^T$, and $\mathbf{K}$ is the matrix with coefficients $K_{ij} = k(\mathbf{x}_i, \mathbf{x}_j)$. Calculating the kernel for two input entries $(\mathbf{x}_i,\, \mathbf{x}_j)$ is equivalent to calculating the dot product of the vectors $(\phi(\mathbf{x}_i),\, \phi(\mathbf{x}_j))$ where ϕ is a *mapping function* which transforms an input vector to a vector of features in a high-dimensional *feature space*. Thus, Equation (6.2) is equivalent to solving the following problem:

$$\boldsymbol{\alpha} = \underset{\boldsymbol{\gamma}}{\text{argmin}}\, \left\|\mathbf{y} - \boldsymbol{\Phi}^T \boldsymbol{\Phi}\, \boldsymbol{\gamma}\right\|^2 \ , \tag{6.3}$$

where $\boldsymbol{\Phi}$ is the matrix that concatenates the vectors of features corresponding to every input entry, $\boldsymbol{\Phi} = [\phi(\mathbf{x}_1), \phi(\mathbf{x}_2), \ldots, \phi(\mathbf{x}_N)]$. Note that the function ϕ is implicit, as it is determined by the choice of a kernel, and it is never calculated in practice. This property is often referred to as the "kernel trick" (Cortes and Vapnik, 1995).

6.2.2 Kernel-recursive least-squares

The Kernel-recursive least-squares (KRLS) algorithm (Engel, Mannor, and Meir, 2004) is an online algorithm which computes an approximate solution to Equation (6.3). The main advantage of KRLS is that the complexity of the obtained prediction model does not depend directly on the size of the dataset, but rather on how complex the dataset is. This is made possible by the fact that the training entries may be linearly dependent in the feature space, *i.e.*, matrix $\mathbf{\Phi}$ can be approximated as:

$$\mathbf{\Phi} \approx \tilde{\mathbf{\Phi}}\,\mathbf{A}^T\ , \tag{6.4}$$

where $\tilde{\mathbf{\Phi}}$ consists of a subset of the columns of $\mathbf{\Phi}$ and matrix $\mathbf{A}$ expresses the columns of $\mathbf{\Phi}$ as linear combinations of the columns of $\tilde{\mathbf{\Phi}}$. In other words, KRLS selects a subset of input entries and computes the prediction model coefficients, $\tilde{\boldsymbol{\alpha}}$, defined by:

$$\tilde{\boldsymbol{\alpha}} = \operatorname*{argmin}_{\gamma} \left\| \mathbf{y} - \mathbf{\Phi}^T\tilde{\mathbf{\Phi}}\,\gamma \right\|^2 = \operatorname*{argmin}_{\gamma} \left\| \mathbf{y} - \mathbf{A}^T\tilde{\mathbf{K}}\,\gamma \right\|^2\ , \tag{6.5}$$

where $\tilde{\mathbf{K}}$ is the matrix of the kernel values for the subset of training entries. The solution to Equation (6.5) is given by: $\tilde{\boldsymbol{\alpha}} = \tilde{\mathbf{K}}^{-1}(\mathbf{A}^T\mathbf{A})^{-1}\mathbf{A}^T\mathbf{y}$.

In the KRLS algorithm the approximation defined by Equation (6.4) is controlled by a parameter, ν: the larger ν, the smaller the dictionary and the larger the error. As in previous chapters, the dictionary refers to a subset of of training examples which are used to define the model. The approximation error made on matrix $\mathbf{K}$ can be expressed as:

$$\mathbf{K} = \mathbf{A}\tilde{\mathbf{K}}\mathbf{A}^T + \mathbf{R}\ \ , \tag{6.6}$$

where $\mathbf{R}$ is a matrix of residual errors. Engel, Mannor, and Meir (2004) showed that the l_2 norm of $\mathbf{R}$ is bounded by $N\nu$, where the l_2 norm of a matrix is defined as:

$$\|\mathbf{R}\|_2 = \max_{\mathbf{u}:\|\mathbf{u}\|_2=1} \|\mathbf{R}\mathbf{u}\|_2\ \ . \tag{6.7}$$

6.2.3 Related Work

Other works related to the problem of parallelising kernel based learning algorithms can be loosely placed into two categories: (1) The data is partitioned and submodels are created, which are later combined into a single model; (2) a distributed approach is taken to construct a single model.

In general, works which take the approach of creating submodels, such as Graf et al. (2004), Yang (2006), Lu, Roychowdhury, and Vandenberghe (2008), Caruana, Li, and Qi (2011), Caruana, Li, and Liu (2013) and Lu, Wang, and Wen (2004), achieve a high speedup over batch algorithms and have little I/O overhead between nodes. However, this approach often has the following drawbacks: degradation in model accuracy, particularly as the number of processing nodes increases (Caruana, Li, and Qi, 2011; Caruana, Li, and Liu, 2013; Lu, Wang, and Wen, 2004); multiple passes of the data are required to guarantee convergence (Graf et al., 2004; Yang, 2006; Lu, Roychowdhury, and Vandenberghe, 2008); and the resultant model is much larger than one resulting from a batch approach (Lu, Wang, and Wen, 2004). To our knowledge, none of these methods provide a guarantee of the accuracy of the model for a single pass of the data.

A notable example of a work which creates a single distributed model is the paper by Chang (2011). The authors report a large speedup over LIBSVM (Chang and Lin, 2011). The main drawback of this approach is the communication overhead, which accounts for over 50% of the running time on some datasets for a large number of machines.

There has been substantial work in this area regarding both the combination of models and parallel training (Bhaduri et al., 2008). The important distinction between the two lies in their methodology; the combination of models implies training several submodels, whereas parallel training methods take a distributed approach to constructing a single model.

Due to its popularity, the Support Vector Machine (SVM) has been the focus of these two methods. In 2004, Graf et al. (2004) proposed a parallel SVM methodology called the Cascade SVM (CSVM). Similar to the method presented in this work, it operates by constructing a tree structure that trains and passes support vectors down to the next layer. This process continues until a single set of support vectors and weights are found. Using this approach, a significant speedup can be achieved by reducing the active set size of each processing node. While convergence to a global optimum is guaranteed, multiple passes of the data is required for this to be achieved. The authors suggest that for many problems only a single pass of the data is required. However, a bound on the error for a single pass is not provided.

Yang (2006) made modifications to CSVM, extending it to use multiple different classifiers, and improving the accuracy over the standard CSVM for classification. The feedback topology is also modified to improve the accuracy at each iteration. The authors report a speedup of up to 40% when comparing to the standard cascade SVM.

Taking a different approach, Chang (2011) presented a body of work that implemented a methodology where several SVMs are trained across multiple nodes. Each node creates an SVM model on a subset of the training data. During the submodel training, once an SVM model finds a support vector (analogous to a dictionary entry for KRLS), then that support vector is distributed amongst all other machines and the process is restarted. This is repeated until all support vectors have been found. The memory load on each node is reduced by efficiently calculating an approximate factorisation of the kernel matrix as an initial step. The authors report $169\times$ speedup over LIBSVM (Chang and Lin, 2011) when running on 500 machines. The main drawbacks of the methodology are the synchronisation and communication overheads, which account for over 50% of the running time on some datasets for a large number of machines.

Alham et. al. have presented two papers, one focusing on the distribution of the SMO algorithms (Alham et al., 2011), and the second looking at a parallel bagging approach (Alham et al., 2013). The first paper introduced MapReduce SMO (MRSMO), which applied several SMO modules to sub sets of the overall dataset. MapReduce Ensemble SVM (MRESVM) was presented in the second paper, focused on a parallel bagging approach where the results from several predictors are combined to created a weighted result of all predictors. MRESVM was tested in both experimental and simulation environments and achieved significantly reduced training time and high levels of accuracy.

Caruana, Li, and Qi (2011) and Caruana, Li, and Liu (2013) presented two papers that developed and tested a parallel SMO training methodology based on MapReduce. The methodology involves several SMO modules which identify support vectors to train on. This reduces the overall training time significantly and uses ontology based concepts. A speedup of $70\times$ is reported using 5 nodes, however, convergence is not guaranteed. The drawback in this approach is that there is a degradation in accuracy as the number of splits in the data increases, the authors report an increase in classification error of up to $1.7\times$ for splits of up to 48.

Lu, Roychowdhury, and Vandenberghe (2008) extended the ideas from Chang (2011) and Yang (2006) and developed a Distributed Parallel SVM (DPSVM). The DPSVM is designed to exchange support vectors, obtained from training multiple sub-modules in a strongly connected network. The methodology presented offered several different architectures, one of which was analogous to the cascade SVM. Lu, Roychowdhury, and Vandenberghe (2008) provide performance results over a number of different topologies/sizes showing that several of their proposed architectures scale well on network sizes of up to 15 nodes. Again, several iterations over the dataset are required for convergence and only theoretically proved in a strongly connected network.

Lu, Wang, and Wen (2004) developed the Min-Max Modular SVM (M3-SVM) which is based on a similar technique used in neural networks (Lu and Ito, 1999). The M3-SVM is used to combine the models created by SVM classifiers which is architecturally similar to the cascade SVM. The authors report up to $4\times$ and $2\times$ speedup over the standard SVM and cascade SVM respectively. However, the approach does degrade the accuracy of the model and up to $2\times$ more support vectors are found which increases the computational complexity for future predictions.

Alternating Direction Method of Multipliers (ADMM) (Boyd et al., 2011) has been gaining popularity in recent years. ADMM can be applied to many optimisation problems, including SVM (Forero, Cano, and Giannakis, 2010; Suzuki, 2013), which allows them to be implemented using parallel and distributed processors with very little inter-node communication.

6.3 Distributed KRLS learning

In the case where KRLS is used on large training data sets, it may be advantageous to divide the learning operation into multiple processes running in parallel. The training data set is divided into subsets, $\mathbf{X} = [\mathbf{X_1}, ..., \mathbf{X_K}]$ and $\mathbf{y} = [\mathbf{y_1}^T, ..., \mathbf{y_K}^T]^T$, and sent to individual computation nodes for processing. Each node creates a model represented by a dictionary $\mathcal{D}_k$, and the corresponding weights, $\boldsymbol{\alpha}_k$. These models must then be combined with each other to form a unique model representing the whole training data set. In this section we show how multiple KRLS models can be combined so that the data contained in the entire training set is optimally approximated.

6.3.1 Concatenating KRLS models

The simplest way to combine KRLS models is to form a dictionary which concatenates the dictionary entries from every individual model, *i.e.*, $\bar{\mathcal{D}} = [\mathcal{D}_1, \ \ldots, \ \mathcal{D}_K]$. A new set of weights corresponding to dictionary $\bar{\mathcal{D}}$, $\bar{\boldsymbol{\alpha}}$, must then be calculated so that the prediction error is minimal for the entire training data set. In other words, denoting $\bar{\mathbf{\Phi}}$ as the dictionary $\bar{\mathcal{D}}$ mapped into the feature space, we have:

$$\bar{\boldsymbol{\alpha}} = \underset{\boldsymbol{\gamma}}{\operatorname{argmin}} \left\| \mathbf{y} - \mathbf{\Phi}^T \bar{\mathbf{\Phi}} \boldsymbol{\gamma} \right\| \approx \underset{\boldsymbol{\gamma}}{\operatorname{argmin}} \left\| \mathbf{y} - \bar{\mathbf{A}} \bar{\mathbf{\Phi}}^T \bar{\mathbf{\Phi}} \boldsymbol{\gamma} \right\| \ , \tag{6.8}$$

where $\bar{\mathbf{A}}$ is the diagonal-block matrix comprised of the $\mathbf{A}$ matrices for every KRLS model, $\bar{\mathbf{A}} = \operatorname{diag}\left([\mathbf{A}_1, \ldots, \mathbf{A}_K]\right)$. The closed-form solution for $\bar{\boldsymbol{\alpha}}$ is therefore given by:

$$\bar{\boldsymbol{\alpha}} = \left(\bar{\mathbf{\Phi}}^T \bar{\mathbf{\Phi}}\right)^{-1} \left(\bar{\mathbf{A}}^T \bar{\mathbf{A}}\right)^{-1} \bar{\mathbf{A}}^T \mathbf{y} \, . \tag{6.9}$$

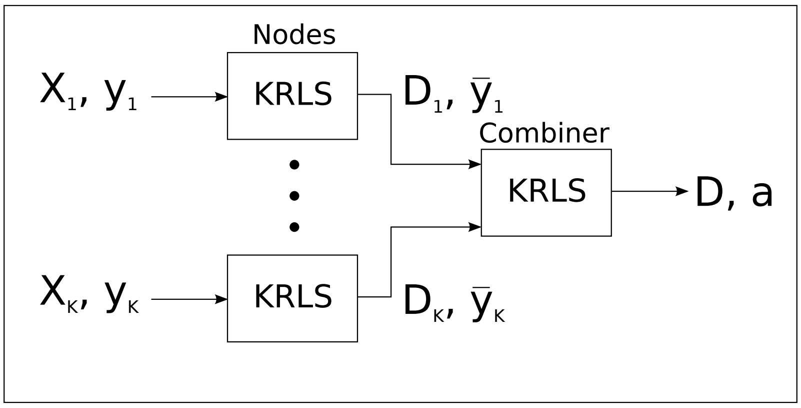


FIGURE 6.1: High level view of the DistKRLS learning method.

We now show that the weights $\bar{\boldsymbol{\alpha}}$ can be calculated without using the entire training data set, $\mathbf{y}$. In other words, $\mathbf{y}$ does not have to be stored after the parallel training stage. Using the block-diagonal structure of $\bar{\mathbf{A}}$, Equation (6.9) can be rewritten as:

$$\bar{\boldsymbol{\alpha}} = \left(\bar{\boldsymbol{\Phi}}^T \bar{\boldsymbol{\Phi}}\right)^{-1} \bar{\mathbf{y}} \,. \tag{6.10}$$

$$\text{where } \bar{\mathbf{y}} = \left[\hat{\mathbf{y}}_1^T, \ldots, \hat{\mathbf{y}}_K^T\right]^T \text{ and } \hat{\mathbf{y}}_k = \left(\mathbf{A}_k^T \mathbf{A}_k\right)^{-1} \mathbf{A}_k^T \mathbf{y}_k \,.$$

Recalling that the weights obtained for an individual model are given by $\boldsymbol{\alpha}_k = \left(\boldsymbol{\Phi}_k^T \boldsymbol{\Phi}_k\right)^{-1} \left(\mathbf{A}_k^T \mathbf{A}_k\right)^{-1} \mathbf{A}_k^T \mathbf{y}_k$, we have

$$\hat{\mathbf{y}}_k = \left(\boldsymbol{\Phi}_k^T \boldsymbol{\Phi}_k\right) \boldsymbol{\alpha}_k \,, \tag{6.11}$$

where $\boldsymbol{\Phi}_k$ denotes the dictionary $\mathcal{D}_k$ mapped in the feature space. Therefore, the coefficients of $\bar{\mathbf{y}}$ simply correspond to the data predicted by applying each KRLS model to its dictionary entries.

6.3.2 Combining KRLS models using the KRLS algorithm

From Equation (6.10), we see that $\bar{\boldsymbol{\alpha}}$ is the solution to the following least-square problem:

$$\bar{\boldsymbol{\alpha}} = \underset{\boldsymbol{\gamma}}{\operatorname{argmin}} \left\| \bar{\mathbf{y}} - \bar{\boldsymbol{\Phi}}^T \bar{\boldsymbol{\Phi}} \boldsymbol{\gamma} \right\|^2 = \underset{\boldsymbol{\gamma}}{\operatorname{argmin}} \left\| \bar{\mathbf{y}} - \bar{\mathbf{K}} \boldsymbol{\gamma} \right\|^2 \,. \tag{6.12}$$

Comparing this equation with Equation (6.3), we observe that this problem is equivalent to estimating a model for the training set data comprised of: a) the entries of the concatenated dictionary, $\bar{\mathcal{D}}$; and b) the vector $\bar{\mathbf{y}}$. Therefore, the KRLS algorithm can be used to estimate this model. This leads to the distributed KRLS (DistKRLS) learning method summarised in Figure 6.1:

1. the original training data set is divided into chunks that are sent to K parallel computing nodes;
2. the KRLS algorithm is used to derive a prediction model for each data chunk;
3. a new training data set is formed of the dictionary $\bar{\mathcal{D}}$ and vector $\bar{\mathbf{y}}$; and
4. the KRLS algorithm is used to estimate a model for the new data set.

An advantage of this approach, is that it can leverage existing infrastructure used to implement MapReduce (Dean and Ghemawat, 2008), such as Hadoop.[1] Algorithm 6.1 outlines the DistKRLS submodel training stage, which can be used as a map function as part of a MapReduce framework. This map function is computed on each computational unit, but with a different subsection of data. This algorithm illustrates how the KRLS module first learns on the training set followed by the evaluation of its *labels*, L, by predicting on its dictionary, $\mathcal{D}$, and weights, $\boldsymbol{\alpha}$.

Algorithm 6.1 DistKRLS map function

```
procedure MAP($\mathbf{X}_{train}, \mathbf{y}_{labels}$)
    $[\mathcal{D}, \boldsymbol{\alpha}] \leftarrow KRLS(\mathbf{X}_{train}, \mathbf{y}_{labels})$ % Train the submodel.
    $L \leftarrow Predict_{\mathcal{D},\boldsymbol{\alpha}}(\mathcal{D})$ % Re-evaluate each dictionary entry to make new labels, using Equation (6.1).
    return($\mathcal{D}, L$)
end procedure
```

These labels along with the dictionary are passed on to the final node which then combines them as outlined in Algorithm 6.2. The function described in Algorithm 6.2 which can be used as the reduce function as part of a MapReduce framework. This processes iteratively combines the submodules whilst retaining the compact nature of the dictionary.

Algorithm 6.2 DistKRLS reduce function

```
procedure REDUCE($list(\mathcal{D}_1, ..\mathcal{D}_N), list(L_1, ..., L_N)$)
    for $i$ in $1 : N$ do
        $[\mathcal{D}_c, \boldsymbol{\alpha}_c] \leftarrow KRLS(\mathcal{D}_i, L_i)$
    end for
    return($\mathcal{D}_c, \boldsymbol{\alpha}_c$)
end procedure
```

It is important to keep in mind that KRLS is an online regression problem, hence combining the submodels can be performed iteratively thus allowing the computation of the reduce step to overlap with the computation of the map step. Depending on the dataset size and the number of splits, this could result in significant performance improvements over a batch approach.

Another advantage of this method is that it may lead to a more compact model than that defined by $\bar{\mathcal{D}}$ if there exists redundancies between entries of this dictionary. The drawback however is that discarding some dictionary entries may increase the prediction error. Let us define the solution after a second application of KRLS as $\breve{\boldsymbol{\alpha}}$, with a corresponding compact kernel matrix, $\breve{\mathbf{K}}$, and expansion matrix, $\breve{\mathbf{A}}$. Matrices $\breve{\mathbf{K}}$ and $\breve{\mathbf{A}}$ provide an approximation of $\bar{\mathbf{K}}$, as follows:

$$\bar{\mathbf{K}} = \breve{\mathbf{A}}\breve{\mathbf{K}}\breve{\mathbf{A}}^T + \breve{\mathbf{R}} \ , \tag{6.13}$$

where $\breve{\mathbf{R}}$ is the residual error introduced by the second application of the KRLS algorithm. Similarly, $\bar{\mathbf{K}}$ and $\bar{\mathbf{A}}$ provide an approximation of $\mathbf{K}$ and we can write:

$$\mathbf{K} = \bar{\mathbf{A}}\bar{\mathbf{K}}\bar{\mathbf{A}}^T + \bar{\mathbf{R}} = \bar{\mathbf{A}}(\breve{\mathbf{A}}\breve{\mathbf{K}}\breve{\mathbf{A}}^T + \breve{\mathbf{R}})\bar{\mathbf{A}}^T + \bar{\mathbf{R}} \ . \tag{6.14}$$

[1]https://hadoop.apache.org/

TABLE 6.1: Summary of the operations required to update a KRLS model based on a new example

Algorithm	KRLS
exp	$\tilde{N}$
$\times$	$3\tilde{N}^2 + M\tilde{N} + 5\tilde{N} + \sum_{i=1}^{\tilde{N}} i$
$+$	$3\tilde{N}^2 + 3M\tilde{N} + \sum_{i=1}^{\tilde{N}} i$
$\div$	1
$<$	1
Total	$6\tilde{N}^2 + 4M\tilde{N} + 6\tilde{N} + 2 + 2\sum_{i+1}^{\tilde{N}} i$
Mem. (input)	$(M+1)w$
Mem. (model)	$(\tilde{N}M + \tilde{N} + 2\sum_{i=1}^{\tilde{N}} i)w$
Mem. (update)	$(\tilde{N} + \sum_{i=1}^{\tilde{N}} i)w$
Intensity as $\tilde{N} \to \infty$	$4.7/w$
Intensity as $M \to \infty$	$4/(\tilde{N}+1)w$
Intensity as $\tilde{N}, M \to \infty$	$4.4/w$

Thus, the l_2 norm of the total residual error, $\mathbf{R}_T$, in the representation of $\mathbf{K}$ is:

$$\begin{aligned}\|\mathbf{R}_T\|_2 &= \left\|\bar{\mathbf{R}} + \bar{\mathbf{A}}\breve{\mathbf{R}}\bar{\mathbf{A}}^T\right\|_2 \\ &\leq N\nu + \psi\bar{N}\nu \ , \end{aligned} \tag{6.15}$$

where ψ is the maximum singular value of $\bar{\mathbf{A}}^T\bar{\mathbf{A}}$, $\bar{N}$ is the length of $\bar{\mathbf{y}}$ and the second line invokes the bound of the error introduced by the KRLS algorithm (Engel, Mannor, and Meir, 2004). Interestingly, since $\bar{\mathbf{A}}$ has block diagonal structure, Equation (6.15) can be rearranged into a form that is easy to calculate.

$$\begin{aligned}\psi_{max}(\bar{\mathbf{A}}^T\bar{\mathbf{A}}) &= \max(\psi_{max}(\mathbf{A}_1^T\mathbf{A}_1), ..., \psi_{max}(\mathbf{A}_K^T\mathbf{A}_K)) \\ &= 1/\min(\psi_{min}(\mathbf{P}_1), ..., \psi_{min}(\mathbf{P}_K)) \ , \end{aligned} \tag{6.16}$$

where $\psi_{max}(B)$, $\psi_{min}(B)$ are the largest / smallest singular values of B respectively. The last line in Equation (6.16) utilises the fact that $\mathbf{P_k} = (\mathbf{A_k}^T\mathbf{A_k})^{-1}$ (Engel, Mannor, and Meir, 2004). Since $\mathbf{P}_k$ is already stored on the k^{th} mapper, after training if an error bound is required, each mapper can calculate $\psi_{min}(\mathbf{P}_k)$ and send the result to the combiner with a constant I/O overhead. The theoretical error bound on real datasets in studied further in Section 6.4.

6.3.3 Performance Modelling

Considering Algorithms 6.1 and 6.2 and Table 3.2, a simple performance model for DistKRLS can be made. For convenience, we repeat the KRLS column from Table 3.2 in Table 6.1. Table 6.1 shows the required number of arithmetic operations, memory reads/writes and arithmetic intensity (number of arithmetic operations per byte of memory read/written) for KRLS to update its model, given a new training example. Note, that $\tilde{N}$ refers to the number of entries in the dictionary, $\mathcal{D}$, and w is the wordlength (in bytes) of the datatype used for memory storage. In terms of arithmetic operations, we can see that the worst-case scalability for KRLS is $O(\tilde{N}^2 + M\tilde{N})$

where $\tilde{N}^2$ will dominate if $\tilde{N} > M$, or $M\tilde{N}$ will dominate if $\tilde{N} < M$. Using the worst-case scalability, for standard KRLS we can estimate the total time taken to be $t = N\tau(\tilde{N}^2 + M\tilde{N})$, where τ is a positive constant. From this simple model, we can estimate the computational time for DistKRLS to be:

$$t_K = \frac{N}{K}\tau(\tilde{N}^2 + M\tilde{N}) + K\tilde{N}\tau(\tilde{N}^2 + M\tilde{N}) \ , \tag{6.17}$$

where the first term is the time taken for each submodel to train (in parallel) on it's subset of training data and the second term is the time taken for the reducer KRLS algorithm to combine the submodels. Here, we assume that each submodel has a model size that is the same size as the resultant model size, $\tilde{N}$. In reality, the actual submodel size is likely to be smaller than this. Defining $\hat{\tau} = \tau(\tilde{N}^2 + M\tilde{N})$ we can rewrite Equation (6.17) as:

$$t_K = \hat{\tau}(\frac{N}{K} + K\tilde{N}) \ . \tag{6.18}$$

So with this model, there is a balance between the time taken for the submodels to train and the time taken for the combination model to train. This is complicated by the complexity of the model, parameterised by the dictionary size, $\tilde{N}$. If K is very high, then the first term takes very little amount of time, while the second term may take a vast amount of time. Conversely if K is small, the submodel training will take a significant amount of time, while the model combining time will be very small. Taking this performance model, we can do performance simulations, by simply specifying N, K and $\tilde{N}$. Figure 6.2 shows the normalised execution time estimate of DistKRLS using Equation (6.18) with $N = 10^6$ and varying K and $\tilde{N}$. In Figure 6.2 when $K = 1$, this refers to regular KRLS. We can see that as K increases from $K = 1$, at first there is a significant drop in estimated execution time, as the first term in Equation (6.18) rapidly decreases. After this initial performance improvement, continuing to increase K increases the execution time until it approaches the execution time of KRLS. This is from the second term of Equation (6.18) increasing as K increases further. In an ideal case for improving performance, it is clear that if N and $\tilde{N}$ are known, we can find an optimal K to minimise Equation (6.18) as follows:

$$K_O = \underset{K}{\operatorname{argmin}} \left(\hat{\tau}(\frac{N}{K} + K\tilde{N}) \right) \ , \tag{6.19}$$

where K_O is the optimal value for K. This can be solved directly by finding the derivative of t_K with respect to K:

$$\frac{dt_K}{dK} = \hat{\tau}\left(\frac{-N}{K^2} + \tilde{N}\right) \ , \tag{6.20}$$

and find K_O for when $\frac{dt_K}{dK} = 0$:

$$0 = \hat{\tau}\left(\frac{-N}{K_O^2} + \tilde{N}\right) \tag{6.21}$$

$$K_O = \sqrt{\frac{N}{\tilde{N}}} \ . \tag{6.22}$$

For $N = 10^6$ and varying $\tilde{N}$ as 10, 100, 1000 and 10000, this gives us the values of K_O as 316, 100, 32 and 10 respectively. This correlates perfectly with what is shown in Figure 6.2. Figure 6.2 also shows that the performance improvement is largely

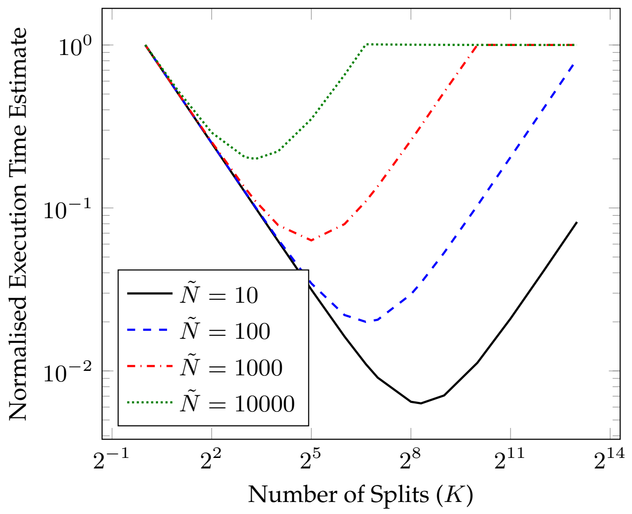


FIGURE 6.2: Normalised execution time estimate of DistKRLS while varying K and $\tilde{N}$

dependent on $\tilde{N}$ with speedups between 10-100$\times$ being achievable. However, this model does not take into account communication overheads, as such, caution should be taken before drawing any conclusions from this analysis. Furthermore, when training on a real dataset, $\tilde{N}$ won't be known a-priori. To estimate it, we suggest training on a random subset of training data until $\tilde{N}$ saturates.

6.4 Results

In this section, the accuracy and performance of the DistKRLS learning algorithm is shown. Synthetic benchmarks for regression and classification are used for training and test data. For comparison, results for the non-parallel KRLS (henceforth referred to as batch KRLS) algorithm is provided along with SVM and cascade support vector machine (CSVM) (Graf et al., 2004), the latter of which is a technique to implement SVM on a distributed computing platform. CSVM was implemented in a standard binary tree configuration and was only allowed a single pass of the data. Readers should note that if multiple passes were allowed, CSVM would converge to the batch SVM solution but would suffer a performance penalty, which would likely be a linear increase in execution time.

6.4.1 Accuracy

In order to demonstrate the modelling accuracy of DistKRLS, the algorithm is tested on both a regression and a classification benchmark. DistKRLS and CSVM were tested while varying the number of *splits* in the training data. If $splits = 1$, then this refers to either batch KRLS or SVM. Otherwise, splits refers either to the number of submodels created by DistKRLS, or to the number of submodels created at the first layer of CSVM.

For the regression test, the Mackey-Glass Chaotic Time Series (Mackey, Glass, et al., 1977) (MG) was used with the chaotic parameter set to 30. The data was configured for a single step prediction problem with a time embedding of 7. In this test, 20 datasets were generated. Each dataset size was 2×10^5, 20% of the data in each set was used as a test set. For all learning algorithms, the Gaussian kernel was

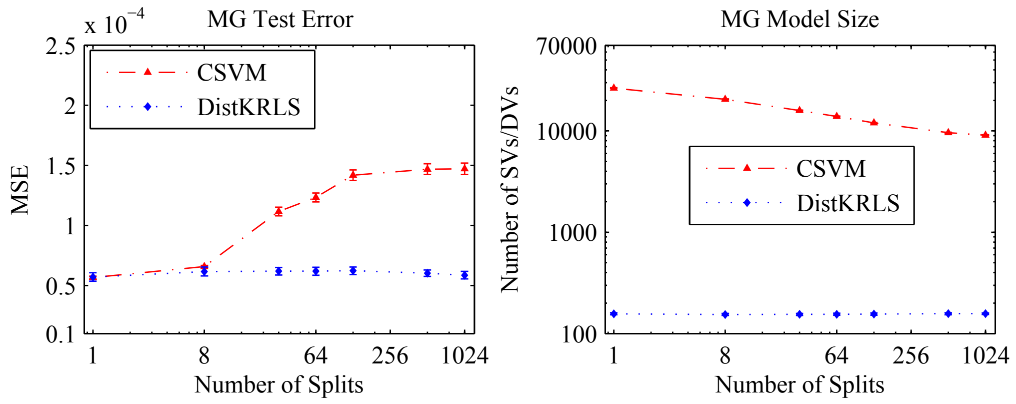


FIGURE 6.3: Mean squared error (MSE) on the test set and model size for each algorithm.

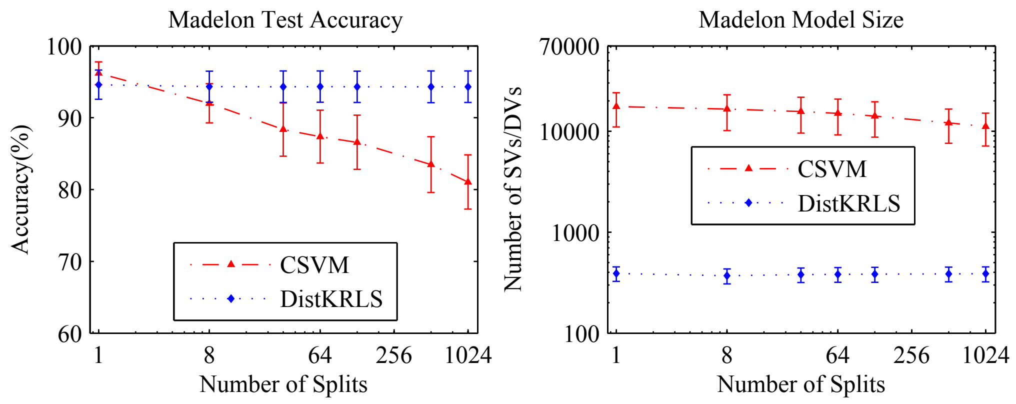


FIGURE 6.4: Classification accuracy on the test set and model size.

used, $\kappa(\mathbf{x}_i, \mathbf{x}_j) = \exp(-\gamma\|\mathbf{x}_i - \mathbf{x}_j\|^2)$, with $\gamma = 0.5$. The configuration for batch and DistKRLS was $\nu = 10^{-4}$. SVM and CSVM used ϵ-SVR (Vapnik, 2000), provided by LIBSVM (Chang and Lin, 2011), with $\epsilon = 0.01$. The same parameters were used for DistKRLS and CSVM as their batch counterparts. Figure 6.3 shows the average mean squared error (MSE) and model size for each algorithm across the 20 sets. The error bars denote one standard deviation above and below the average. Note that "model size" refers to the number of support vectors or dictionary vectors found by the algorithm. Clearly, DistKRLS and batch KRLS perform almost identically, while the accuracy of CSVM degrades slightly with increasing numbers of splits. Specifically, the maximum increase in average MSE between all configurations of DistKRLS and KRLS was 9.09%, while for CSVM it was 160% when compared to SVM. Also, the average model size of KRLS and DistKRLS is more than $150\times$ smaller. This also means the computational cost of subsequent predictions is over $150\times$ smaller.

For the classification test, the Madelon Classification Data (Guyon et al., 2004) (MAD) was used. In this test, 40 datasets were generated. Each set contained 2×10^5 examples and was generated with 4 informative features and 4 redundant features, and 20% of the data in each set was used as the test set. The Madelon input data was normalised prior to training/prediction, and each output label was either -1 or 1. The exponential kernel was used again with $\gamma = 0.5$. KRLS and DistKRLS was used with $\nu = 0.7$. SVM and CSVM used C-SVC (Cortes and Vapnik, 1995), also provided by LIBSVM (Chang and Lin, 2011), with $C = 50$. Figure 6.4 shows the classification accuracy and model size between DistKRLS and CSVM over the 40 sets. The error bars

Worst Case Error Bounds (ΔMSE)		
	MG	MAD
Batch	1.0e-4	1.0e-4
8	1.3e-2	3.1e-2
128	1.1e-2	5.9e-3
512	2.8e-3	3.8e-3
1024	2.5e-2	1.1e-2
16384	2.6e-4	2.0e-4
32768	2.0e-4	2.0e-4

FIGURE 6.5: Worst case error bounds on the Mackey-Glass and Madelon datasets.

denote one standard deviation above and below the average. Using the classification data, the difference between our approach and CSVM is much more pronounced. While SVM works very well, the accuracy of CSVM deteriorates significantly while requiring over $40\times$ more support vectors than the KRLS approaches. Specifically, the greatest difference in average classification accuracy between DistKRLS and KRLS was 0.30%, compared with 15.1% between CSVM and SVM.

Error Bounds

In this subsection the worst case error bounds are considered, which are calculated using Equation (6.16). Comparisons with actual error measured in the previous section are also provided. Figure 6.5 shows the calculated worst case error bounds on several real datasets.

In order to understand how the change in MSE, ΔMSE, impacts on the accuracy of the model the relative error should be considered. Figure 6.6 shows the relative theoretical and measured ΔMSE using the models created by the Madelon dataset. The relative measured ΔMSE is calculated as follows: $\Delta MSE_m = (MSE_c - MSE_b)/MSE_b$, where MSE_c is the calculated MSE of the proposed method and MSE_b is the calculated MSE of the batch model. The relative theoretical error is calculated using: $\Delta MSE_t = (\nu_m + \sqrt{\Psi(\bar{\mathbf{A}}^2)\nu_c} + \nu_b - MSE_b)/MSE_b$, where $\Psi(\bar{\mathbf{A}}^2)$ is calculated using Equation (6.16). Note that when tested using 512 nodes, the measured error rises above the theoretical error bound. We suspect that this is due to floating point rounding errors. As such, floating point errors in KRLS based algorithms will be studied in future work. Other than this single point, where the number of nodes is 512, the measured change in MSE validates the theoretical error bound.

6.4.2 Performance

In this section, the execution time of the algorithms is considered. The Madelon dataset experiments were run on a Ubuntu Linux system with 2x Intel(R) Xeon(R) E5506 CPUs at 2.13GHz and 48GB of RAM. The Mackey-Glass experiments were run on a 16 node cluster running Centos Linux. MATLAB was used to run all tests, however, LIBSVM was used to implement all SVM algorithms and a C library was created to implement all KRLS algorithms.[2] All linear algebra routines used in the KRLS C code linked back to MATLAB's BLAS library, which provided the highest performance when compared with ATLAS (Whaley and Petitet, 2005) and

[2]Software available at: `https://bitbucket.org/nick_fraser/libkaf/`

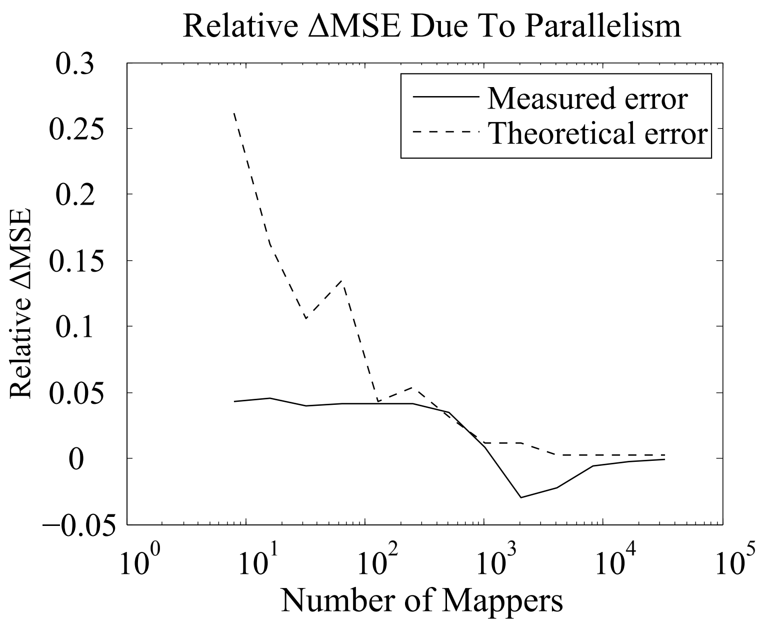


FIGURE 6.6: A comparison between a worst case error bound and measured error bound using the Madelon dataset.

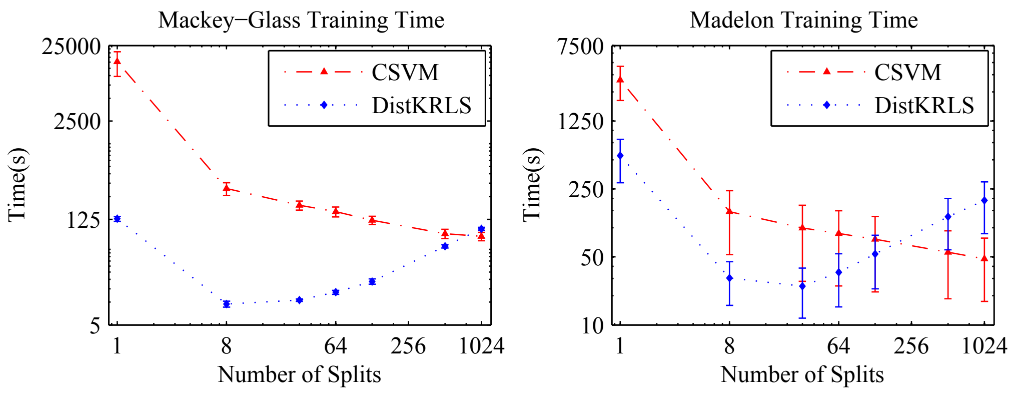


FIGURE 6.7: Training times of CSVM and DistKRLS will varying K

OpenBLAS (Wang et al., 2013). MATLAB's parallel computing toolbox was used to provide parallelism for the DistKRLS and CSVM.

Figure 6.7 shows the training time for each of the different algorithms. CSVM achieves a significant performance increase over SVM which continues to steadily improve with an increasing number of splits. DistKRLS performs best for either 8 or 32 splits, but approaches batch KRLS as the number of splits increases. This is due to an increase in computation time for combining the KRLS submodels. Compared to batch KRLS, the best average speedup achieved by DistKRLS is 22.7×. In all but 3 configurations, DistKRLS has lower average execution time than CSVM. Based on the results, it is likely that a continued increase in splits would cause CSVM to execute faster than DistKRLS.

It should also be noted that the DistKRLS training time curves shown in Figure 6.7 correlate very well with the curves shown in Figure 6.2. Furthermore, given the model sizes shown in Figure 6.6, we can estimate the expected optimal number of splits, K_O, using Equation (6.22). For the Mackey-Glass times series, $K_O \approx 30$, which is slightly higher than the optimal point shown in Figure 6.7, but still corresponds to a near-optimal point on the performance curve. For the Madelon dataset, $K_O \approx 20$, which lies right in-between the two most optimal points which were tested. This shows that while the model in Section 6.3.3 is simple, it can be effective for estimating good values for K for a given dataset and hyperparameter settings for DistKRLS.

6.5 Extensions

In this section, we look at some extensions to enhance the DistKRLS algorithm. Specifically, we look at extending the model to a generalised, multi-layered tree structure. Afterwards, we look at making some modifications to KRLS itself which aim to improve the error bound, given in Equation (6.15).

6.5.1 Generalised Tree Structure

In this subsection, the error bound related to a generalised tree structure is considered. Starting with Equation (6.15) and the error bound of the KRLS algorithm. Figure 6.8

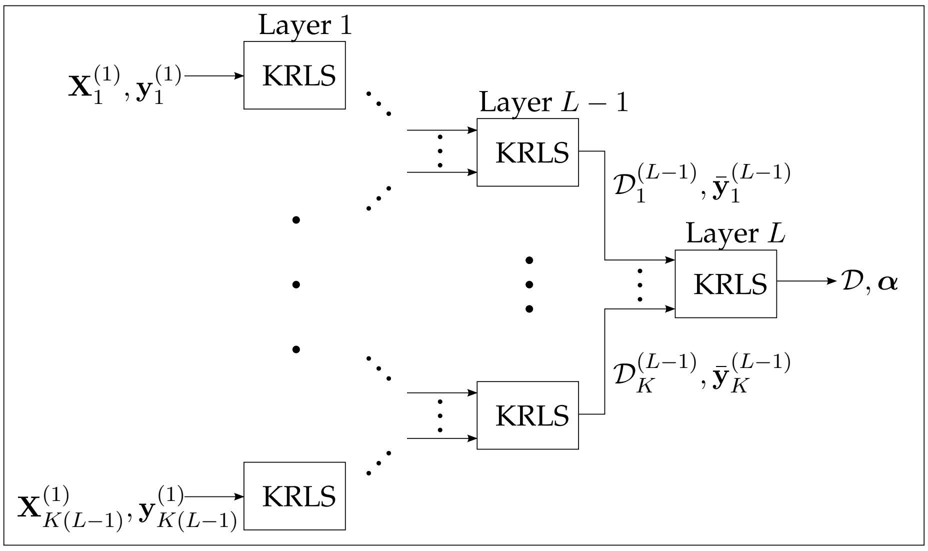

Layer 1
$\mathbf{X}_1^{(1)}, \mathbf{y}_1^{(1)}$
KRLS
Layer $L-1$
KRLS
$\mathcal{D}_1^{(L-1)}, \bar{\mathbf{y}}_1^{(L-1)}$
Layer L
KRLS
$\mathcal{D}, \boldsymbol{\alpha}$
$\mathcal{D}_K^{(L-1)}, \bar{\mathbf{y}}_K^{(L-1)}$
KRLS
$\mathbf{X}_{K(L-1)}^{(1)}, \mathbf{y}_{K(L-1)}^{(1)}$
KRLS

FIGURE 6.8: A generalised version of DistKRLS which uses a k-nary tree.

shows a general DistKRLS tree with L layers, constructed using a k-nary tree of size K. Clearly, the result of a standard two layer DistKRLS implementation is another KRLS model which increases error bounds on the representation of $\mathbf{K}$. As such, there's no reason that two or more separate parallel DistKRLS could be computed and combined using the same method described in Section 6.3. Applying such logic recursively results in a general version of DistKRLS which can have some arbitrary tree structure at the cost of some possible further increases in error bounds. In this section, we'll use the same notation as Section 6.2.2, with the inclusion of superscripts and subscripts denoting the layer number and split number respectively. For example, $\mathcal{D}_k^{(l)}$ denotes the dictionary found by the k^{th} KRLS module in the l^{th} layer of the DistKRLS tree. Recalling Equation (6.13), the relation between the original kernel matrix and the approximated kernel matrix, becomes:

$$\mathbf{K} = \bar{\mathbf{A}}^{(1)}\bar{\mathbf{K}}^{(1)}\bar{\mathbf{A}}^{(1)T} + \bar{\mathbf{R}}^{(1)} = \bar{\mathbf{A}}^{(1)}(\bar{\mathbf{A}}^{(2)}\bar{\mathbf{K}}^{(2)}\bar{\mathbf{A}}^{(2)T} + \bar{\mathbf{R}}^{(2)})\bar{\mathbf{A}}^{(1)T} + \bar{\mathbf{R}}^{(1)T} \quad , \tag{6.23}$$

Similarly, Equation (6.15), showing the total residual error, becomes:

$$\begin{aligned}\|\mathcal{R}\|_2 &= \left\|\bar{\mathbf{R}}^{(1)} + \bar{\mathbf{A}}^{(1)}\bar{\mathbf{R}}^{(2)}\bar{\mathbf{A}}^{(1)T}\right\|_2 \\ &\leq N^{(1)}\nu^{(1)} + \psi^{(1)}N^{(2)}\nu^{(2)} \quad , \end{aligned} \tag{6.24}$$

where $\mathcal{R}$ is the total residual error in the representation of $\mathbf{K}$ using a 2-layer DistKRLS tree and $\psi^{(1)}$ is the largest singular value of $\bar{\mathbf{A}}^{(1)T}\bar{\mathbf{A}}^{(1)}$. Generalising Equation (6.24) to a tree with L layers becomes:

$$\begin{aligned}\|\mathcal{R}\|_2 &\leq N^{(1)}\nu^{(1)} + \psi^{(1)}N^{(2)}\nu^{(2)} + \cdots + N^{(L)}\nu^{(L)} \prod_{n=1}^{(L-1)} \psi^{(n)} \\ &\leq \sum_{l=1}^{L} N^{(l)}\nu^{(l)} \prod_{n=1}^{(l-1)} \psi^{(n)} \quad . \end{aligned} \tag{6.25}$$

This shows that for the cost of potentially further increasing the maximum error, DistKRLS can be extended to a generalised tree structure which allows one to exploit further parallelism that may be available in certain systems. This structure may be particularly useful when the time taken for the combiner is significantly longer than the time taken for each of the nodes, as we can further parallelise the combiner into sub-DistKRLS steps. Alternatively, in scenarios with sparsely connected network topologies this generalised tree structure would help to reduce the burden on network bandwidth.

6.5.2 The Kernel Recursive Least Squares Submodel

In this subsection, we look at modifying the KRLS algorithm itself to improve the error bound, as stated in Equation (6.25). Firstly, let us assume that we would like the worst case error bound in DistKRLS to be less than or equal to the error bound of KRLS. The first obvious thing to notice, is that if $\nu^{(2)} \to \nu^{(L)} = 0$, then the residual error in the representation of $\mathbf{K}$ is equivalent to KRLS. Clearly, this would provide a similar worst case error bound to KRLS, but, would likely produce a much larger model than one found simply by KRLS. We assume a large model (i.e. one with many dictionary entries) is undesirable due to the increased cost (linear with the number of dictionary entries) of making predictions. As such, we assume that a kernel regression model produced this way would be non-ideal.

Modify the Approximate Linear Dependency Condition

As stated in Section 6.2.2, the KRLS algorithm tries to approximate $\mathbf{\Phi}$ using a compact subset. The result is stated in Equation (6.4). The KRLS algorithm calculates each row in $\mathbf{A}$ by seeing if each example satisfies the ALD condition: $\delta_t \leq \nu$, where

$$\delta_t = \min_{\mathbf{a}} \left\| \sum_{j=1}^{\tilde{N}_t} a_j \phi(\tilde{\mathbf{x}}_j) - \phi(\mathbf{x}_t) \right\|^2 \quad , \tag{6.26}$$

where $\mathbf{a} = [a_1, ..., a_{\tilde{N}_t}]$, $\mathbf{a}$ becomes the t^{th} row in $\mathbf{A}$ and $\tilde{N}_t$ refers to the number of entires in $\mathcal{D}$ at time t. Clearly, using this minimisation function does not place any constraint on $\|\mathbf{a}\|_2$. This means that prior to training, one cannot place a bound on ψ and therefore, one cannot calculate a worst case bound on Equation (6.26). An obvious improvement to this bound then, is to somehow place a bound on $\|a\|_2 < \epsilon$, where $\epsilon \in \mathbb{R}_{>0}$. Then to add this condition to the ALD condition, meaning a new example is *not* added to the dictionary if $\delta_t < \nu$ && $\|a\|_2 < \epsilon$. This will in turn introduce a bound on $\psi_{max}(\mathbf{A}_k^T\mathbf{A}_k)$, which can further simplify Equation (6.16). With the introduction of this extra condition on ALD, $\psi_{max}(\mathbf{A}_k^T\mathbf{A}_k)$ must fall within the following bound:

$$\psi_{max}(\mathbf{A}_k^T\mathbf{A}_k) \leq \frac{N}{K}\epsilon \ , \tag{6.27}$$

Meaning that Equation (6.15) can be updated as follows:

$$\begin{aligned}\|\mathbf{R}_T\|_2 &= \left\|\bar{\mathbf{R}} + \bar{\mathbf{A}}\breve{\mathbf{R}}\bar{\mathbf{A}}^T\right\|_2 \\ &\leq N\nu + \psi\bar{N}\nu \\ &\leq N\nu + \frac{N}{K}\epsilon\bar{N}\nu \ ,\end{aligned} \tag{6.28}$$

This means the maximum error bounds can be determined a-priori, at hyperparameter setting time, rather than calculated during training as Equation (6.16) suggests. We can also now use this formulation to update Equation (6.25) as follows:

$$\begin{aligned}\|\mathcal{R}\|_2 &\leq N^{(1)}\nu^{(1)} + \psi^{(1)}N^{(2)}\nu^{(2)} + \cdots + N^{(L)}\nu^{(L)}\prod_{n=1}^{(L-1)}\psi^{(n)} \\ &\leq \sum_{l=1}^{L} N^{(l)}\nu^{(l)}\prod_{n=1}^{(l-1)}\psi^{(n)} \\ &\leq \sum_{l=1}^{L} N^{(l)}\nu^{(l)}\prod_{n=1}^{(l-1)}\frac{N^{(n+1)}}{K^{(n)}}\epsilon^{(n)} \ .\end{aligned} \tag{6.29}$$

6.6 Conclusion

This chapter has presented a new method of combining multiple models of kernel based regression algorithms. This technique removes the dependency between models for the bulk of the data at training time which allows it to be distributed among many machines. In this work, the KRLS algorithm is used for training as it provides a compact, near optimal least squares model of the training data. The KRLS algorithm also provides a parameter which trades off model accuracy with computational time. Utilising only the KRLS algorithm for model training and combining, we show that the residual MSE is bounded in the final model based on the properties of the data and the ALD threshold. To the best of our knowledge, the algorithm proposed provides the only distributed/parallel kernel based regression algorithm capable of providing a bounded approximation error while only requiring a single pass through the data. In terms of performance, even on a single machine with 8 cores, a speedup of $22\times$ was achieved when compared to a batch execution of the KRLS algorithm. Compared to another method of parallelising kernel algorithms, CSVM, DistKRLS achieves higher performance (for smaller dataset splits), but most importantly maintains high accuracy for single passes through the dataset. The method described, thanks to the KRLS algorithm, also returns a far more compact model than the other distributed techniques. This has the two-pronged effect of greatly reducing the I/O between the mappers and the combiner, and also providing a model with much smaller computation complexity for future predictions.

We have shown that with a modification to the ALD criterion, a-priori error bounds can be placed on the reconstruction error at hyperparameter setting time. Finally, we have also shown that DistKRLS can be applied to generalised tree structures, while still maintaining bounds on the worst-case reconstruction error.

In future work, the approximation error bound will be explored on real data sets. The processing and I/O time achieved on real clusters, and massively parallel processors (GPUs/FPGAs) will be tested. The accuracy of the algorithm using several distribution and sorting schemes will be considered. Also, the accuracy of the technique using other types of KAFs as mapping functions will be tested, such as stochastic gradient descent algorithms which offer an extra reduction in computational cost.

Chapter 7

Conclusion

In this thesis, we have demonstrated that algorithmic (Chapter 5 and Chapter 6) modifications and custom compute architectures (Chapter 4 and Chapter 6) can significantly improve the performance of implementations of KAFs. In particular, we show that large scale models of traditional KAFs are entirely memory bound (Chapter 3), while small models, which can fit in device on-chip memory, are compute bound. We show that the precision requirements associated with KAFs vary quite significantly depending on whether a particular KAF calculates an optimal, least squares model at each update or whether it tends towards the global minima using a SGD. In particular, we show that SGD based models are much more amenable to quantisation.

Extending this, we show that if we want to train many small scale models, such as when performing hyperparameter optimisation, we can achieve very high performance using custom architectures on FPGAs. Furthermore, for application requirement with extremely high data-rate requirements, in the order of 100s of MHz, that these can be attained through delayed model adaptation and custom architectures capable of fine-grained parallelism.

Finally, in order to extend these benefits to large scale models, we show that large models can be decomposed into several smaller scale models and later combined to form a more complex model. Unlike prior works, this is achievable without either:

- causing significant accuracy degradation; or
- requiring multiple iterations through the dataset.

Putting this together, we've shown:

- that many independent small scale KAF models can be accelerated;
- that large models can be created by combining several smaller independent KAF models; and
- that high data rate applications can be addressed with a simple dependency management technique.

As such, we've shown that for a range of application requirements, through the use of custom compute architectures and algorithmic modifications, that KAFs can be applied to broad range of vastly different applications.

Our hope is that hardware engineers or machine learning engineers are able to realise these possibilities and that this opens up KAFs to a much broader array of applications that weren't previously considered, like those discussed in Section 2.2.4.

7.1 Future Work

In this work, we targeted a broad set of applications with varying requirements. As such, the evidence that we gathered for any particular application domain is limited. Clearly, future work is to show that the results demonstrated in this work generalise to more KAF algorithms. For example, QKLMS is a promising SGD based KAF which behaves a little differently to KNLMS. Exploring whether delayed model adaptation works on QKLMS and other SGD based KAFs would be very valuable work, as it would demonstrate these techniques are more broadly applicable than what is demonstrated in this thesis.

Furthermore, implementing pipelined, parallel, independent versions of RLS based KAFs using deeply pipelined architectures, such as those suggested in Chapter 4 would be very valuable work. As the storage requirements, and the computational patterns of RLS based KAFs are significantly different to SGD based KAFs, this may be non-trivial and would also provide valuable insights.

Similarly, it would be interesting to see if the dependency issues which are overcome for SGD based KAFs can be applied to RLS based KAFs. Again, the different computational patterns would make this task non-trivial.

Given that many machine learning algorithms use SGD as their core optimisation method, such as logistic regression and deep neural networks, it would be interesting to see if the techniques described in Chapters 4 and 5 would also apply to those families of algorithms. The application of SGD in such algorithms means that similar data dependencies would appear to ones described in these chapters, as many algorithms based on SGD have similar compute patterns to KAFs like KNLMS.

Finally, a system which utilises the arbitrary tree structure available in DistKRLS, but each node executes multiple, smaller scale KRLS submodels, i.e., using the techniques of Chapter 4 before combining models locally on the device and then globally, would require significant technical competence spanning multiple application domains and be a significant contribution.

Glossary

AA affine arithmetic. 35, 125

AI arithmetic intensity. 3, 41, 42, 44–46, 125

ALD approximate linear dependency. 25, 106, 119–121, 125

ALD-KRLS approximate linear dependency kernel recursive least squares. 14, 15, 93, 94, 125

ALU arithmetic logic unit. 125

ATLAS Automatically Tuned Linear Algebra Software. 100, 125

AWS Amazon Web Sevices. 40, 125

AXI advanced extensible interface. 100, 125

BGD batch gradient descent. 6, 8, 18, 125

BNN binarised neural network. xxi, 3, 125, 129–134, 137, 139, 141

CESTAC contrôle et estemation stochastique des arrondis calculs. 35, 36, 125

CFG control flow graph. 35, 125

CNN convolutional neural network. 125, 129–131, 134, 135, 138, 139

CPU central processing unit. ix, xvii, 2, 27, 28, 33, 37, 40, 42, 45, 46, 50, 125

CSVM cascade support vector machine. xviii, 114–117, 120, 125

DistKRLS distributed kernel recursive least squares. xviii, 2, 112–121, 124, 125

DKNLMS delayed kernel normalised least mean squares. xviii, xxi, 2, 71, 72, 76, 78, 79, 84, 85, 89–100, 103, 125

DKNLMS-CT delayed kernel normalised least mean squares with correction terms. xviii, 2, 71, 81, 83–85, 88, 91–93, 95, 96, 125

DKNLMS-DG delayed kernel normalised least mean squares with dictionary guarding. xviii, 2, 71, 81, 82, 85, 87, 91–95, 98, 125

DLMS delayed least mean squares. 75, 76, 79, 125

DSL domain specific language. 89, 125

DSP digital signal processor. 41, 46, 90, 98, 99, 125

FB-KRLS fixed budget kernel recursive least squares. 24, 76, 101, 125

FIFO first in, first out queue. 100, 125

FIR finite impulse response. 30, 125

FPGA field programmable gate array. v, xiii, xvii, 3, 5, 27–29, 33, 37, 40, 42, 45–47, 49, 50, 71, 75, 76, 90, 91, 100, 103, 104, 121, 123, 125, 129–131, 133, 138, 139, 141

FPS frames per second. 125, 134

FSM finite state machine. 35, 125

FWHT fast Walsh-Hadamard transform. 76, 125

GFLOP billion floating point operations. 125

GFLOPS billion floating point operations per second. 41, 125, 129

GOps gigaoperations. 125, 139

GOps/s gigaoperations per second. 41, 42, 45, 101, 103, 125

GPU graphics processing unit. ix, xvii, 2, 27, 37, 40, 42, 45, 46, 50, 121, 125

HLS High-Level Synthesis. 103, 125, 131

I/O input/output. 26, 120, 121, 125

IA interval arithmetic. 35, 125

IFM input feature map. 125, 134

II initiation interval. 125

IIR infinite impulse response. 30, 125

ILSVRC ImageNet Large Scale Visual Recognition Competition. 125

KAF kernel adaptive filter. v, 1–3, 5, 20, 24, 26, 29, 39, 42–48, 50, 71–73, 75, 76, 81, 84, 91, 100, 103, 104, 121, 123–126

KAFBOX the KAF toolbox. 91, 125

KLMS kernel least mean squares. 20, 21, 23, 24, 75, 102, 125

KNLMS kernel normalised least mean squares. xiv, xvii, xviii, 2, 43–50, 71–79, 81, 83, 85, 90–96, 98, 100–103, 124, 125

KRLS kernel recursive least squares. xxi, 2, 14, 20–25, 30, 43–46, 106, 112–120, 124, 125

KRLS-T kernel recursive least squares tracker. 24, 125

LMS least mean squares. 18–21, 30, 47, 48, 71, 72, 75, 94, 125

LOOCV leave-one-out cross-validation. 16, 125

LS-SVM least-squares support vector machine. 25, 27, 125

LUT lookup table. 46, 99, 125

MAC multiply accumulate. 41, 125

MCA Monte Carlo arithmetic. 2, 33, 35, 36, 47–50, 125

MDKNLMS multi-delayed kernel normalised least mean squares. xvii, xviii, 2, 71, 78–81, 83, 85, 91–103, 125

MLP multilayer perceptron. 125, 138, 139

MSE mean squared error. 15, 17, 48, 49, 92, 93, 116, 120, 125

MVTU Matrix–Vector–Threshold Unit. 125, 132, 135, 140

NN neural network. 125

NORMA naive online regularised risk minimisation algorithm. 76, 102, 103, 125

OCM on-chip memory. 125, 129, 131–135, 139

OFM output feature map. 125, 132, 136

OS operating system. 29, 125

PCI Peripheral Component Interconnect. 125

PCI-X Peripheral Component Interconnect Extended. 29, 125

PE Processing Element. 76, 125, 132, 134

PU Pooling Unit. 125

QKLMS quantised kernel least mean squares. 43–46, 124, 125

RLS recursive least squares. 18–22, 24, 30, 48, 71, 72, 101, 124, 125

RTL register transfer level. 89, 125

SGD stocastic gradient descent. 6, 18, 27, 46, 50, 123–125

SIMD single instruction multiple data. 28, 125

SMO sequential minimal optimisation. 25, 125

SVM support vector machine. xiii, xvii, 24, 25, 27, 28, 114–117, 125

SW-KRLS sliding window kernel recursive least squares. xvii, 20, 24, 25, 47–50, 76, 101, 102, 125

SWU Sliding Window Unit. 125, 132, 134

TDP thermal design power. 40, 125

TF-DLMS transpose-form delayed least mean squares. 75, 125

TF-RDLMS transpose-form retimed delayed least mean squares. 75, 125

TOPS trillion operations per second. 125, 129, 130, 138, 139, 141

TU Thresholding Unit. 125

Appendix A

Scaling Binarised Neural Networks on Reconfigurable Logic

A.1 Introduction

Convolutional neural networks (CNNs) provide impressive classification accuracy in a number of application domains, but at the expense of large compute and memory requirements (LeCun et al., 1998). A significant body of research is investigating compression techniques combining numerous approaches such as: weight and synapse pruning; data compression techniques such as quantisation, weight sharing and Huffman coding; and reduced precision with fixed point arithmetic (Han, Mao, and Dally, 2015; Iandola et al., 2016; Iandola et al., 2015). Recently, an extreme form of reduced precision networks, known as BNNs (Courbariaux and Bengio, 2016), have gained significant interest as they can be implemented for inference at a much reduced hardware cost. This is due to the fact that multipliers and accumulators become XNORs and popcounts respectively, and both are significantly lighter in regards to resource and power footprint. For example, a KU115 offers 483 billion floating point operations per second (GFLOPS) compared to 46 trillion operations per second (TOPS) for binary synaptic operations. This is visualised in the roofline models in Figure A.5 which illustrates theoretical peak performance for numerous reduced precision compute operations.[1] Furthermore, the model size is greatly reduced and typically small enough to fit in on-chip memory (OCM), again reducing power, simplifying the implementation and providing much greater bandwidth.

FINN (Umuroglu et al., 2017) describes a framework for mapping BNNs to reconfigurable logic. However, it focuses on BNNs for embedded applications and as such, the results reported are for smaller network sizes running on an embedded platform. In this work, we briefly summarise FINN and analyse it from the perspective of scaling to larger networks and devices, such as those targeted for data centers. Firstly, we focus on several technical issues that arise when scaling networks on FINN including: BRAM usage, throughput limitations and resource overheads. We also identify several properties of CNN layers which make them map to FINN more efficiently. Our results, measured on an ADM-PCIE-8K5 platform (*ADM-PCIE-8K5 Datasheet* 2016), show that indeed very high image classification rates, minimal latency with very high power efficiency can be achieved by mapping BNNs to FPGAs, even though improvements may be made. Secondly, we highlight an issue of padding, a common feature of large CNNs, which may cause significant hardware overheads. We propose an alternative form of padding, which maps more efficiently to reconfigurable logic. Specifically, the contributions of this work are: 1) measured performance results

[1]Assuming 70% device utilisation, 250 MHz clock frequency and 178 LUTs and 2 DSPs per average floating point operation, and 2.5 LUTs per binary XNOR-popcount operation.

for large-scale networks on an ADM-PCIE-8K5 board; 2) an analysis of FINN for large-scale problems, highlighting some bottlenecks as well as proposing solutions; and 3) a form of padding, which achieves high accuracy while also maintaining a binary datapath.

A.2 Background

A great deal of prior work on mapping neural networks to hardware exist for FPGAs, GPUs and ASICs to help increase inference rate or improve energy efficiency. We refer the reader to the work by Misra and Saha (2010) for a comprehensive survey of prior works. In general we distinguish four basic architectures:

1. a *single processing engine*, usually in the form of a *systolic array*, which processes each layer sequentially (Ovtcharov et al., 2015; Zhang et al., 2015; Chen, Emer, and Sze, 2016; Andri et al., 2016);

2. a *streaming architecture* (Venieris and Bouganis, 2016; Alemdar et al., 2017), consisting of one processing engine per network layer;

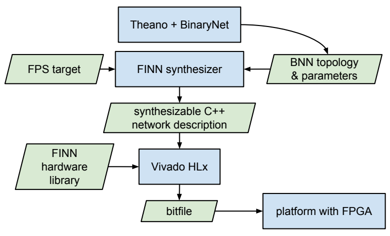
3. a *vector processor* (Farabet et al., 2009) with instructions specific to accelerating the primitives operations of convolutions; and

4. a *neurosynaptic processor* (Esser et al., 2016), which implements many digital neurons and their interconnecting weights.

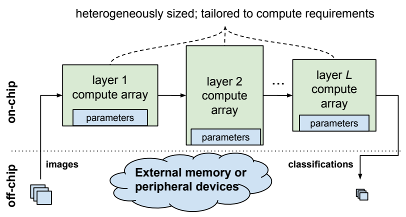
Significant research investigates binarisation of neural networks whereby either input activations, synapse weights or output activations or a combination thereof are binarised. If all three components are binary, we refer to this as *full binarisation* (Kim and Smaragdis, 2016). If not all three components are binary, we refer to this as *partial binarisation*. The seminal XNOR-Net work by Rastegari et al. (2016) applies convolutional BNNs on the ImageNet dataset with topologies inspired by AlexNet, ResNet and GoogLeNet, reporting top-1 accuracies of up to 51.2% for full binarisation and 65.5% for partial binarisation. DoReFa-Net by Zhou et al. (2016) explores reduced precision with partial and full binarisation on the SVHN and ImageNet datasets, including best-case ImageNet top-1 accuracies of 43% for full and 53% for partial binarisation. Finally, the work by Courbariaux and Bengio (2016) describes how to train fully-connected and convolutional networks with full binarisation and batch normalisation layers, reporting competitive accuracy on the MNIST, SVHN and CIFAR-10 datasets. All BNNs used in this work are trained by a methodology based on the one described by Courbariaux and Bengio (2016), and unset bits represent a numerical -1 value while set bits represent a +1. The downside to the high performance characteristics of BNNs is a small drop in accuracy, in comparison to floating point networks. Improving the accuracy for reduced precision CNNs is an active research area in the machine learning community and first evidence shows that accuracy can be improved by increasing network sizes (Sung, Shin, and Hwang, 2015).

A.3 BNNs on Reconfigurable Logic

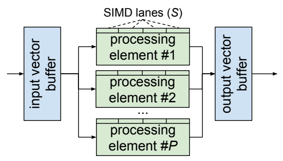
This work builds on top of FINN (Umuroglu et al., 2017), a framework for building scalable and fast BNN inference accelerators on FPGAs. FINN is motivated by observations on how FPGAs can achieve performance in the TOPS range using



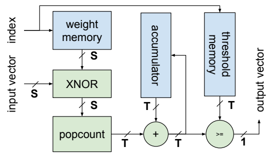
(A) Accelerator generation.



(B) Top-level architecture.



(C) Building block (MVTU).



(D) MVTU datapath.

FIGURE A.1: FINN workflow and architecture, reproduced from Umuroglu et al. (2017).

XNOR–popcount–threshold datapaths to implement the BNNs described by Courbariaux and Bengio (2016). Given a trained BNN and target frame rates, FINN follows the workflow in Figure A.1a to compose a BNN accelerator from hardware building blocks. In more detail, a given network topology and model retrieved through Theano (Theano Development Team, 2016), together with design targets in form of resource availability and classifcation rate, is processed by the synthesiser which determines the scaling settings and produces a synthesisable C++ description of a heterogeneous streaming architecture.[2] The top-level architecture is exemplified in Figure A.1b and has two key differentiators compared to prior work on FPGA CNN accelerators. First, all BNN parameters are kept in OCM, which greatly increases arithmetic intensity, reduces power and simplifies the design. Furthermore, one streaming compute engine is instantiated per layer, with resources tailored to fit each layer's compute requirements and the user-defined frame rate. Compute engines communicate via on-chip data streams and each produces and consumes data in the same order with the aim of minimising buffer requirements in between layers. Thereby each engine starts to compute as soon as the previous engine starts to produce output. In essence, we build a custom architecture for a given topology rather than scheduling operations on top of a fixed architecture, as would be the case for typical systolic array based architectures, and avoid the "one-size-fits-all" inefficiencies and reap more of the benefits of reconfigurable computing.

[2]To achieve portability, we chose a commercial high level synthesis tool, Vivado HLS (Xilinx, 2016), for the implementation. The tool enables faster development cycles via high-level abstractions, and provides automated pipelining to meet the clock frequency target.

A.3.1 The Matrix–Vector–Threshold Unit

In more detail, the key processing engine in FINN is the Matrix–Vector–Threshold Unit (MVTU) as illustrated in Figure A.1c, which computes binarised matrix-vector products and compares against a threshold to generate a binarised activation. Convolutions are *lowered* (Chellapilla, Puri, and Simard, 2006) to matrix–matrix multiplications, using a Sliding Window Unit (SWU) (described further in Appendix A.4.2) to generate the image matrix and the MVTU to carry out the actual arithmetic. The SWU generates the same vectors as those described by Chellapilla, Puri, and Simard (2006), but with the elements of the vector interleaved to reduce and simplify memory accesses and to avoid the need for data transposition between layers. Internally, the MVTU consists of an input and output buffer, and an array of P PEs, shown in Figure A.1d, each with a number of SIMD lanes, S. The synapse weight matrix to be used is kept in OCM distributed between PEs, and the input images stream through the MVTU as each one is multiplied with the matrix. Each PE receives exactly the same control signals and input vector data, but multiply-accumulates the input with a different part of the matrix. A PE can be thought of as a hardware neuron capable of processing S synapses per clock cycle. Finally, the MVTU architectural template can also support partial binarisation for non-binarised outputs and inputs. Removing the thresholding stage provides non-binarised outputs, while using regular multiply-add instead of XNOR-popcount can handle non-binarised inputs. These features are used in the first and last layers of networks that process non-binary input images or do not output a one-hot classification vector.

A.3.2 Folding

Depending on the use case, a neural network inference accelerator may have different throughput requirements in terms of the images classified per second (FPS). In FINN, FPS is controlled by the per-layer parameters P (number of PEs in an MVTU) and S (number of SIMD lanes in each PE). If the number of synapses, Y, connected to a neuron is greater than S, then the computation is *folded* across the PE, with the resulting PE producing an activation every $F^s = Y/S$ clock cycles. Similarly, if the number of neurons, X, in a layer exceeds P, then each PE is responsible for calculating activations for $F^n = X/P$ neurons. In total, it would take the MVTU $F^s \cdot F^n$ clock cycles to compute all its neuron activations. The MVTUs are then rate balanced by adjusting their P and S values to match the number of clock cycles it takes to calculate all required activations for each layer. As this is a balanced streaming system, the classification throughput FPS will be approximately F_{clk}/II, where F_{clk} is the clock frequency, and the II (Initiation Interval) is equal to the total folding factor $F^{\text{tot}} = F^s \cdot F^n$ cycles for a fully-connected layer. Note that convolutional layers have an extra folding factor, F^m, which is the number of matrix–vector products which need to be computed, i.e., the number of pixels in a single output feature map (OFM). Therefore, for convolutional layers the total folding factor is: $F^{\text{tot}} = F^s \cdot F^n \cdot F^m$.

A.3.3 BNN-specific Operator Optimisations

The methodology described by Courbariaux and Bengio (2016) forms the basis for training all BNNs in this work. Firstly, in regards to arithmetic, we are using 1-bit values for all input activations, weights and output activations (full binarisation), where an unset bit represents -1 and a set bit represents +1. Binary dot products result in XNORs with popcounts (which count the number of set bits instead of accumulation with signed arithmetic). Secondly, all BNN layers use batch normalisation (Ioffe and

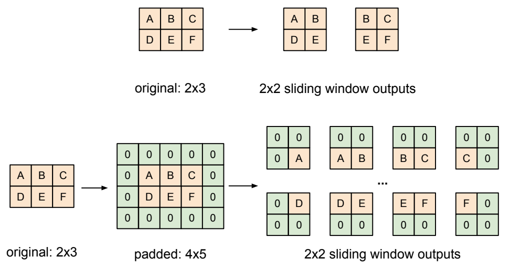


FIGURE A.2: Convolution without (top) and with (bottom) padding.

Szegedy, 2015) on convolutional or fully connected layer outputs, then apply the sign function to determine the output activation. In the work by Umuroglu et al. (2017), it is shown how the same output can be computed via thresholding, which combines the bias term, batch normalisation and activation into a single function. Finally, the networks described by Courbariaux and Bengio (2016) perform pooling prior to activations, i.e. pooling is performed on non-binarised numbers, which are then batch normalised and fed into the activation function. However, as shown by Umuroglu et al. (2017), pooling can be equally performed after activation, once binarised, in which case it can be effectively implemented with the Boolean OR-operator.

A.4 Padding for BNN Convolutions

This section describes the improvements made to FINN in this work.

A.4.1 Padding using nonzero values

Zero-padding is commonly applied for convolutional layers in deep neural networks, in order to prevent the pixel information on the image borders from being "washed away" too quickly (*CS231n: Convolutional Neural Networks for Visual Recognition*). Figure A.2 illustrates the sliding window outputs on the same image with and without padding. Observe that the pixels on the border (such as A and F) occur more frequently in the sliding window outputs when padding is used, thus preventing them from being "washed away" too quickly in the next layer.

A challenge arises for zero-padding in the context of BNNs with only $\{-1, +1\}$ arithmetic: there is no zero value defined. In fact, the original BinaryNet (Courbariaux and Bengio, 2016) paper uses ternary values $\{-1, 0, +1\}$ for the forward pass, with zeros used for padding. However, ternary values require two bits of storage, essentially doubling the OCM required to store values and the bitwidth of the datapath. Since FINN focuses on BNNs that fit entirely into on-chip memory of a single FPGA, minimising the resource footprint is essential. Thus, a padding solution that avoids ternary values is preferable. A straightforward solution would be to use e.g. -1 as the padding value, and expect that the BNN learns weights which compensate for these values. Surprisingly, -1-padding works just as well as 0-padding according to our results, which are presented in Appendix A.5.4.

A.4.2 Streaming padding for FINN

FINN lowers (Chellapilla, Puri, and Simard, 2006) convolutions to matrix-matrix multiplication of the filter weight matrix with the image matrix. The image matrix

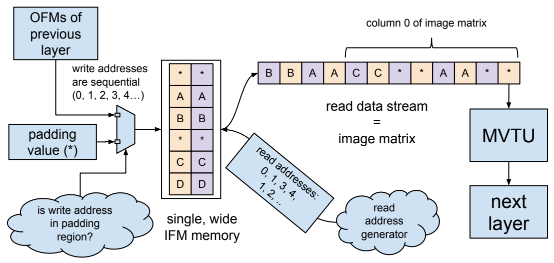

OFMs of previous layer
write addresses are sequential (0, 1, 2, 3, 4…)
padding value (*)
is write address in padding region?
single, wide IFM memory
read addresses: 0, 1, 3, 4, 1, 2, ..
read address generator
column 0 of image matrix
B B A A C C * * A A * *
read data stream = image matrix
MVTU
next layer

FIGURE A.3: FINN SWU enhanced with streaming padding.

is generated on-the-fly by the SWU. Figure A.3 illustrates how the FINN SWU is enhanced to support streaming padding for convolution layers. The key operational principle is the same as in FINN. Namely, a single, wide input feature map (IFM) memory is used to store the feature maps into OCM in the order they arrive, and the addresses that correspond to the sliding window pixels are read out. Padding is achieved by a multiplexer that chooses the data source for writing into the IFM memory. If the current write address falls into the padding region, the padding value (e.g. -1) is written into the memory; otherwise, an element from the output stream of the previous layer is written instead.

A.5 Evaluation

A.5.1 Experimental Setup

BNN Topologies

The network topologies used for our experiments are all based on the CNN topology described in BinaryNet (Courbariaux and Bengio, 2016), which we denote as cnn. This topology is inspired by the VGG16 network (Simonyan and Zisserman, 2014), which consists of three groups of (3x3 convolution – 3x3 convolution – 2x2 maxpooling) layers, and two fully-connected layers at the end. To explore how FINN performs on a range of network sizes, we introduce a scaling factor, σ, to scale the width of each layer, and denote the resulting topology as $\mathrm{cnn}(\sigma)$. Note that σ does not influence the number of layers in a network, it merely affects: 1) the number of neurons in each fully connected layer; and 2) the number of filters in each convolutional layer. Specifically, $\mathrm{cnn}(0.5)$ has half as many filters in each convolutional layer and half as many neurons in each fully connected layer, compared to the CNN described in BinaryNet (Courbariaux and Bengio, 2016). In terms of convolutional networks, FINN (Umuroglu et al., 2017) only evaluated a single non-padded BNN topology ($\mathrm{cnn}_{\mathrm{NoPad}}(^1/_2)$). In this work, we consider $\mathrm{cnn}(^1/_2)$ as well as smaller ($\mathrm{cnn}(^1/_4)$) and bigger ($\mathrm{cnn}(1)$) padded convolutional topologies to investigate how FINN scales.

In order to simulate a realistic use case, we consider an application with a fixed FPS requirement, i.e., real-time object recognition of a video stream. If one considers an 800 $\times$ 600 video stream at 25 FPS, which partitioned into tiles of 32 $\times$ 32 for classification. In order to classify the tiles in real-time, a classification rate of approximately 12 kFPS would be required. We use this image rate as our target for all experiments and adjust the number of PEs and SIMD accordingly in each layer of each design.

TABLE A.1: The expected cost per operation for each precision type.

Datatype	LUTs min	LUTs max	LUTs avg	DSPs min	DSPs max	DSPs avg	C_{avg} $\times 10^{-6}$	C_{rel}
Binary	2.16	4.25	2.90	0	0	0	6.25	1
Int2	5.57	9.95	7.03	0	0	0	15.14	2.42
Int4	13.81	18.89	15.60	0	0	0	33.60	5.38
Int8	42.29	44.78	48.83	0	0	0	105.2	16.84
Int16	10.99	20.38	15.03	0.5	0.5	0.5	90.58	14.50
Float32	178	-	-	2	-	-	383.3	61.38
KU115	-	663,360	-	-	5,520	-	-	-

The Platform

The target board is an Alpha Data ADM-PCIE-8K5 which features a Xilinx Kintex UltraScale XCKU115-2-FLVA1517E FPGA (KU115). The KU115 offers 663k LUTs, 2160 BRAMs (36k) and 5520 DSPs and is running at 125 MHz for our experiments. The host machine is a IBM Power8 8247-21L with 80 cores at 3.69 GHz and 64 GB of RAM and it is running Ubuntu 15.04. In all experiments, all parameters are stored in OCM while the test images and the predicted labels are read from and written to the host memory directly. The provided resource counts include the PCI Express infrastructure used for moving data streams as well as the BNN accelerator. Although we are not able to provide per-experiment power measurements, the maximum power consumption observed for this board was 41 W on a board power dissipation benchmark test, and we expect that the real power dissipation values for BNN accelerators will be significantly lower than this.

A.5.2 Cost of Operational Primitives

In this section, the cost of the fundamental mathematical operations which underpin CNNs are compared. Table A.1 shows the estimated cost per operation for each precision type. Note that for binary networks a operations refer to XNORs & popcounts. For binary operations, the resource cost was estimated by generating multiple versions of the MVTU, as described in Appendix A.3.1. In order to estimate the hardware cost for integer types, a modified version of the MVTU, was created with XNOR replaced with $\times$ and popcount replaced with adder trees. Note that each layer within each adder tree grows in bitwidth to avoid overflow. For both binary and integer types, a number of different MVTUs were created while scaling the input and output vector sizes and the folding factors, F^n and F^s. The numbers in Table A.1 refer to the average area cost/operation across the all of the experiments. In general, MVTUs created with higher F^s incurred a larger hardware overhead. For floating point, the area cost of the individual multiply and addition units were used without constructing MVTUs. The average cost per operation, C_{avg} is calculated as follows:

$$C_{avg} = \max\left(\frac{LUTs/MAC}{LUT_{usage} * LUTs_{TOTAL}}, \frac{DSPs/MAC}{DSP_{usage} * DSPs_{TOTAL}}\right) , \tag{A.1}$$

where $LUTs_{TOTAL}/DSPs_{TOTAL}$ are the total available LUTs/DSPs on the target device and LUT_{usage}/DSP_{usage} are estimates of the proportion of LUTs/DSPs that can be used for arithmetic on the target device. We've estimated $LUT_{usage} = 0.7$ and

TABLE A.2: The accuracy of different sized CNNs on the CIFAR-10 dataset while varying the precision.

Scale Factor	Binary Err. (%)	Int2 Err. (%)	Int4 Err. (%)	Int8 Err. (%)	Int16 Err. (%)	Float Err. (%)	# Params	Ops/frame
0.25	20.94	16.05	14.81	14.29	14.15	15.44	879,914	78,494,720
0.5	14.84	12.39	11.56	11.52	11.62	12.14	3,510,858	310,343,680
1.0	11.67	10.41	9.67	9.69	9.65	10.76	14,025,866	1,234,104,320

$DSP_{usage} = 1.0$ in this work. C_{avg} then becomes the fraction of the target device resources that are used on average by operation for each type. Relative cost, C_{rel}, is used to compare the arithmetic cost of binarised networks against other precision types directly. For example, if a binarised and an Int4 network have been trained to achieve the same level of accuracy, the Int4 network must have 5.38 less operations to have the same accuracy / computation trade-off as the binarised one.[3] Interestingly, modelling computational cost this way means that Int16 has a *lower* hardware cost than Int8, because it uses less LUTs/Op than Int8 and the proportion of DSPs that it uses per Op with respect to the total on the target device, a Xilinx Kintex UltraScale KU115, is less than the proportion of LUTs/Op used by Int8.

A.5.3 Network Size and Accuracy

In this section, we look at the relationship between accuracy, network size and overall hardware cost of a number of different networks and precisions. Specifically, we look at the network topology used by Courbariaux and Bengio (2016) for the CIFAR-10 dataset. We use the same basic network topology, but introduce a scaling factor, which scales the number of OFMs in each convolution layer and the number of neurons in each fully connected layer. The error rates of each network, for each precision type are shown in Table A.2. Note that the accuracy results for the binarised networks utilise padding with a value of -1, rather than 0 and as such, differ from the results reported by Courbariaux and Bengio (2016). For the integer networks, a fixed point representation with a fractional length of $bitwidth - 2$ was used throughout. For activations, the binarised and integer networks utilised the hard tanh function, $tanh_{HARD}(x) = clip(x, -1, 1)$, while the floating point networks used the rectified linear function, $relu(x) = \max(0, x)$, for their activation functions.

Given the data is Tables A.1 and A.2 the error rates of each networks can now be be plotted against the computational cost per image for each precision type. This is achieved by multiplying the Operations / frame of each network configuration by the C_{avg} associated with each precision type. The results are shown in Figure A.4. Also in Figure A.4, is the overall memory size of each network for each precision type. This plot also shows the size of the available BRAMs on the target device. If a particular network configuration requires access to off-chip memory in order to load model parameters, this will likely create a bottleneck and cause the resultant hardware configuration to incur a significant throughput penalty. Both plots can be analysed in two ways: 1) consider points of equal error rate, i.e., a horizontal line through each plot, and find the network configuration and datatype associated with the smallest computational cost or memory size and therefore, the configuration that uses the least resources or can be implemented at the highest framerate on the target device; or 2) consider points of equal computational cost of memory size, i.e., a vertical line through each plot, then find the network type and configuration which

[3] This assumes that both networks have the same memory footprint for their parameters.

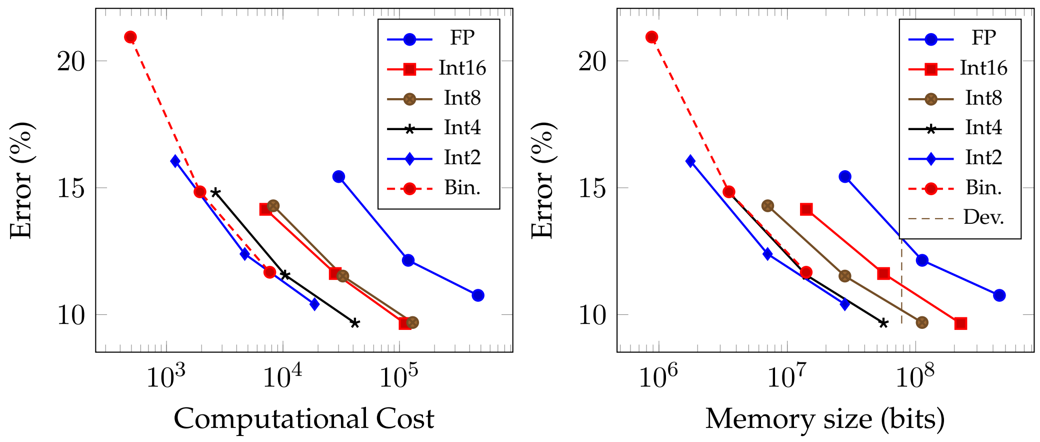


FIGURE A.4: Accuracy versus hardware cost of CNNs with various precision types.

TABLE A.3: Accuracy with different padding modes for CIFAR-10.

		Padding Mode		
		no-padding	0-padding	-1-padding
Scale	$\sigma = {}^1/_4$	75.6%	78.2%	79.1%
	$\sigma = {}^1/_2$	80.1%	85.2%	85.2%
	$\sigma = 1$	84.2%	88.6%	88.3%

corresponds the lowest error rate, the resultant network is the best configuration that can be implemented at given area and throughput constraints. By analysing each plot using these methods, it is clear to see that floating point, Int16 and Int8 networks incur significant resources costs, which are not compensated by a significant enough increase in accuracy to make them worthwhile to implement on the target device. Surprisingly, Binarised, Int2 and Int4 networks have almost equivalent error rate / cost curves (in regions of overlapping error) that is difficult draw conclusions on the best datatype to use for the target device, particularly for such a small sample size. However, given the results that were attained, Int2 appears have slightly better error rate / hardware trade-offs than the other datatypes. Note that the computational cost in Figure A.4 is calculated from the scarcity of the resources available on a Xilinx UltraScale KU115. Therefore, the computation cost is equal to the number of clocks cycles required to infer a single image, given a network configuration and precision type, if all of the available resources (70% LUTs, 100% DSPs) on the device were allocated to arithmetic operations and assuming a 100% scheduling efficiency.

A.5.4 Effects of Padding

To investigate how different padding modes affect accuracy, we trained a set of convolutional BNNs on the CIFAR-10 dataset with different scaling factors (σ). The convolutions used are 3×3, so one pixel of padding is added on each border. The results are summarised in Table A.3. As expected, using 0-padding improves accuracy by 4-5% compared to no-padding, indicating that the conventional wisdom on padding increasing accuracy also applies to BNNs. Furthermore, we can see that the accuracy of -1-padded networks are on par with the 0-padded ones of same scale. This suggests that BNNs are able to learn to compensate for the -1 values used for

TABLE A.4: Operations per image with different padding modes for CIFAR-10.

		Padding Mode		
		no-padding	0-padding	-1-padding
Scale	$\sigma = {}^1/_4$	30.4 M	78.5 M	78.5 M
	$\sigma = {}^1/_2$	118.9 M	310.3 M	310.3 M
	$\sigma = 1$	530.1 M	1234.1 M	1234.1 M

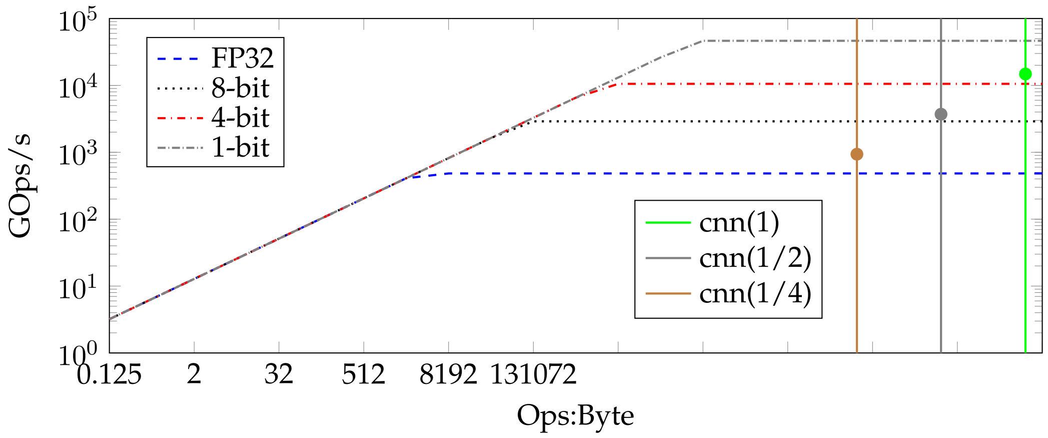


FIGURE A.5: KU115 roofline with different datatypes.

padding by adjusting the weight values and thresholds, and the accuracy benefits can be still obtained with a binary (as opposed to ternary) datapath.

It should also be noted that no-padding results in a significant reduction in the amount of operations per frame and the number of parameters. Thus, it is worthwhile to examine the computation versus accuracy tradeoffs in the context of padding. Table A.4 lists the total number of XNOR-popcount operations necessary to classify one image using different padding modes and scaling factors. We can observe that the no-padding topology variant for the same scale factor requires $2-3\times$ less computation. However, this comes at a cost of higher error rate, and a smaller-but-padded network may be advantageous over a larger-but-not-padded network. For instance, $\mathrm{cnn}({}^1/_4)$ classifies at 79% accuracy using 78.5 M operations, whereas the $\mathrm{cnn}_{\mathrm{NoPad}}({}^1/_2)$ classifies at 80.1% accuracy using 118.9 M operations. Thus, $\mathrm{cnn}({}^1/_4)$ may be preferable due to its lower computational cost if a 1% drop in accuracy is acceptable for the use case at hand.

A.5.5 Scaling to Larger Networks

A results summary is shown in Table A.5 which also shows the accuracy achieved by the implemented networks on a number of benchmark datasets. The new padded CNN results are provided in the top portion of Table A.5, while key results from FINN (Umuroglu et al., 2017) are shown in the lower portion. Note that for comparison, scaled versions of the multilayer perceptrons (MLPs) consisting only of fully-connected layers described in BinaryNet(Courbariaux and Bengio, 2016) are also shown and denoted as $\mathrm{mlp}(\sigma)$.

We can see that larger networks scale well to larger FPGAs, with our best designs achieving 14.8 TOPS and 671 μs image classification latency. Furthermore, even with

TABLE A.5: Key performance and resource utilisation results achieved by this work (top) and FINN (bottom) on a number of BNN topologies.

	Network	Device	LUT	BRAM	kFPS	GOps/s
Padded	cnn($^1/_4$)	KU115	35818	144	12.0	938
	cnn($^1/_2$)	KU115	93755	386	12.0	3,711
	cnn(1)	KU115	392947	1814	12.0	14,814
FINN	$\text{cnn}_{\text{NoPad}}$($^1/_2$)	Z7045	54538	192	21.9	2,466
	mlp($^1/_{16}$)	Z7045	86110	130.5	12,361	8,265
	mlp($^1/_8$)	Z7045	104807	516.5	6,238	11,613
	mlp($^1/_4$)	Z7045	79097	398	1,561	9,086

the largest network tested, all model parameters fit within OCM of the KU115 and thus avoids potential bottlenecks on external memory access. However, if we were to attempt a larger network (such as cnn(2)) the design would no longer fit in OCM without also reducing the frame rate. This is discussed further in Appendix A.5.5.

While the results described in Table A.5 represent state-of-the-art in terms of image classification rates and energy efficiency, it is still work in progress. Our best raw performance number (14.8 TOPS) outperforms that of the smaller FPGA device used in FINN (Umuroglu et al., 2017) (11.6 TOPS), which is no surprise. However, the MLPs shown in FINN (Umuroglu et al., 2017) do achieve performance figures closer to the theoretical peak of the device. This is mostly due to the simplicity of MLPs versus CNNs. Figure A.5 shows the estimated peak performance of the KU115 with vertical lines indicating the arithmetic intensity of the 3 CNN networks and coloured markers indicating actual performance of FINN. We can see that our implementations still fall below the KU115's theoretical peak. We expect that with planned improvements, including those in Appendix A.5.5, significant performance gains can still be achieved. However it should be noted, that the largest design cnn(1) shown in Table A.5 requires 1.2 GOps per frame, which is similar in computational requirements to the popular AlexNet (Krizhevsky, Sutskever, and Hinton, 2012) which requires 1.45 GOps per frame. In comparison the GPUs, the NVidia Titan X can achieve 3.2 kFPS at 227 W for AlexNet inference, compared to 12 kFPS at less than 41 W on the KU115 FPGA.[4] It should be noted that these figures are in terms of 32-bit floating point operations, as opposed to the binarised ones discussed in this work. However, high accuracy has been achieved by fully binarised (Hubara et al., 2016) and partially binarised (Zhou et al., 2016) versions of AlexNet and we expect to be able to achieve high performance on such networks.

BRAM Efficiency

Since FINN currently focuses on BNNs that fit entirely onto the on-chip memory of a single FPGA, making the most out of the available on-chip memory is essential. Figure A.6 illustrates how much of the allocated BRAM space (as reported by Vivado) is actually utilised by the accelerator. The two largest contributors to BRAM usage in FINN are the network parameters (BNN weights and thresholds), and stream buffers (such as FIFOs and input-output buffers), which are shown with different colors in the bar chart. As can be expected, the majority of the utilised storage is for weights,

[4]`https://www.nvidia.com/content/tegra/embedded-systems/pdf/jetson_tx1_whitepaper.pdf`

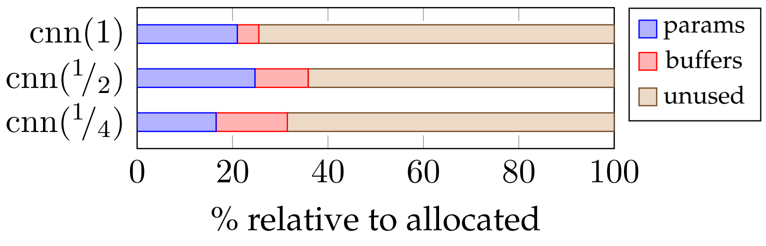


FIGURE A.6: Utilisation of allocated BRAM storage space.

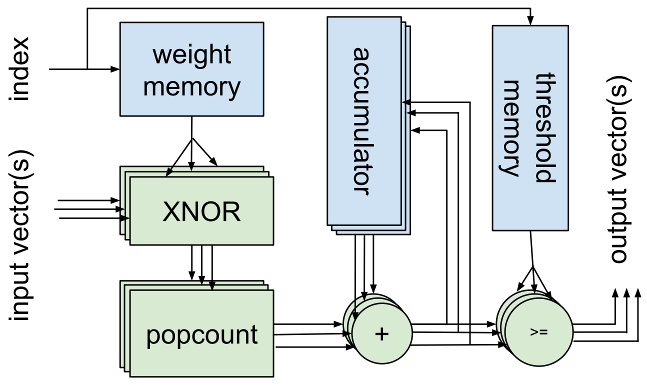


FIGURE A.7: Datapath for matrix–multiple vector product.

although the streaming buffers occupy roughly equal storage for $\mathrm{cnn}(^1/_4)$ since there are not as many parameters.

A bigger concern is that on average only ~22% of the storage space in the allocated BRAMs is actually used. For scaling to even larger networks, this under–utilisation could constitute a problem as synthesis will fail trying to allocate more BRAMs than is available in the FPGA. Further analysis into this issue revealed that this is a consequence of how convolutions are currently handled in FINN. Recall that the total folding factor is $F^{\mathrm{tot}} = F^s \cdot F^n \cdot F^m$ for a convolution layer. The F^m folding factor here arises due to implementing matrix–matrix products as a sequence of matrix–vector products Unlike F^s and F^n, F^m is currently not controllable, since only one matrix–vector product is computed at a time in each MVTU. When high FPS is desired, the initiation interval must be minimised, which can only be achieved by small values F^n and F^s since F^m is constant. This requires creating many PEs and SIMD lanes operating in parallel, each of which have their own weight and threshold memories operating independently. However, this causes the weight matrix to be split and distributed into many small pieces, thus causing the observed storage under–utilisation.

One way of addressing this problem would be enabling control over the F^m parameter by enhancing the MVTU to enable multiplying the same matrix by multiple vectors in parallel. In this manner, fewer PEs and SIMD lanes could be instantiated, each working on a larger portion of the weight matrix and utilising BRAM storage better. Figure A.7 shows how the MVTU datapath could be enhanced to support multiple vectors, broadcasting the same data from the weight memory to multiple XNOR-popcount-accumulate datapaths. Note that only the datapath is duplicated; the weight and threshold memories have a single copy. We leave further investigation of the matrix–multiple vectors for future work.

A.6 Conclusion

In this work, we explored the scaling of BNNs on large FPGAs using the FINN framework. We highlight an issue with padding in convolutional layers in BNNs described in BinaryNet (Courbariaux and Bengio, 2016) which would cause them to require a 2-bit datapath. We show that a small modification to padding (padding with -1 values) improves accuracy over no-padding and is comparable to 0-padding, while still allowing networks to maintain a binary datapath. We found that high performance for large networks can be attained, with our highest demonstrated performance achieving 12 kFPS at less than 41 W of board power and 14.8 TOPS of raw computational performance. When scaling to large networks, we also show that the efficiency of BRAM usage in FINN is low, and propose an architectural modification which would allow for better BRAM utilisation. Alternatively, if a higher number of smaller BRAMs were available on FPGAs devices, this would allow FINN to better exploit the available resources.

For future work, we will further enhance the FINN framework to support partial binarisation, and different kinds of convolutional layers, such as inception layers (Szegedy et al., 2015) and fire-modules (Iandola et al., 2016). The architectural improvements, described in Appendix A.5.5 will be implemented to further improve the BRAM usage efficiency of architectures produced by FINN. Further networks which have been trained on larger datasets, i.e., ImageNet, will also be implemented. Finally, better power measurements will be attained rather than using "worst-case" power dissipation values.

Bibliography

ADM-PCIE-8K5 Datasheet (2016). 1.3. Alpha Data.

Alachiotis, Nikolaos and Alexandros Stamatakis (2011). “FPGA Optimizations for a Pipelined Floating-point Exponential Unit”. In: *Proceedings of theInternational Symposium on Applied Reconfigurable Computing*. Springer, pp. 316–327.

Alemdar, Hande et al. (2017). “Ternary neural networks for resource-efficient AI applications”. In: *Proceedings of the International Joint Conference on Neural Networks*. IEEE, pp. 2547–2554.

Alham, Nasullah Khalid et al. (2011). “A MapReduce-based distributed SVM algorithm for automatic image annotation”. In: *Computers and Mathematics with Applications*. Vol. 62, pp. 2801–2811. DOI: 10.1016/j.camwa.2011.07.046.

Alham, Nasullah Khalid et al. (2013). “A MapReduce-based distributed SVM ensemble for scalable image classification and annotation”. In: *Computers and Mathematics with Applications*. Vol. 66, pp. 1920–1934. ISBN: 978-1-4673-0024-7. DOI: 10.1016/j.camwa.2013.07.015.

Amodei, Dario et al. (2016). “Deep speech 2: End-to-end speech recognition in english and mandarin”. In: *Proceedings of the International Conference on Machine Learning*, pp. 173–182.

Andri, Renzo et al. (2016). “YodaNN: An Ultra-Low Power Convolutional Neural Network Accelerator Based on Binary Weights”. In: *arXiv Computer Research Repository* abs/1606.05487.

Anguita, Davide, Andrea Boni, and Sandro Ridella (2003). “A digital architecture for support vector machines: theory, algorithm, and FPGA implementation”. In: *IEEE Transactions on Neural Networks* 14.5, pp. 993–1009.

Anguita, Davide et al. (2007). “A hardware-friendly support vector machine for embedded automotive applications”. In: *Proceedings of the International Joint Conference on Neural Networks*. IEEE, pp. 1360–1364.

Anguita, Davide et al. (2011). “A FPGA core generator for embedded classification systems”. In: *Journal of Circuits, Systems and Computers* 20.02, pp. 263–282. DOI: 10.1142/S0218126611007244.

Anscombe, Francis J (1973). “Graphs in statistical analysis”. In: *The American Statistician* 27.1, pp. 17–21.

Aronszajn, Nachman (1950). “Theory of Reproducing Kernels”. In: *Transactions of the American mathematical society* 68.3, pp. 337–404.

Athanasopoulos, Andreas et al. (2011). “GPU acceleration for support vector machines”. In: *Procs. 12th Inter. Workshop on Image Analysis for Multimedia Interactive Services (WIAMIS 2011), Delft, Netherlands*.

Avnet (2018). *Ultra96 Hardware User's Guide*. URL: http://www.zedboard.org/sites/default/files/documentations/Ultra96-HW-User-Guide-rev-1-0-V0_9_preliminary.pdf (visited on 02/25/2020).

Baboulin, M. et al. (2009). "Accelerating scientific computations with mixed precision algorithms". In: *Computer Physics Communications* 180.12, pp. 2526–2533.

Bachrach, Jonathan et al. (2012). "Chisel: constructing hardware in a Scala embedded language". In: *Proceedings of the 49th Annual Design Automation Conference*. ACM, pp. 1216–1225.

Bartlett, Peter (2008). *Statistical Learning Theory: Reproducing Kernel Hilbert Spaces*. URL: https://people.eecs.berkeley.edu/~bartlett/courses/281b-sp08/ (visited on 01/12/2020).

Baskin, Chaim et al. (2018a). "Nice: Noise injection and clamping estimation for neural network quantization". In: *arXiv preprint arXiv:1810.00162*.

Baskin, Chaim et al. (2018b). "UNIQ: Uniform Noise Injection for Non-Uniform Quantization of Neural Networks". In: *arXiv preprint arXiv:1804.10969*.

Bellman, Richard E. (1957). *Dynamic Programming*. Princeton University Press. ISBN: 9780691079516.

Bergstra, James and Yoshua Bengio (2012a). "Random search for hyper-parameter optimization". In: *J. Mach. Learn. Res.* 13, pp. 281–305.

— (2012b). "Random search for hyper-parameter optimization". In: *J. Mach. Learn. Res.* 13, pp. 281–305.

Bhaduri, Kanishka et al. (2008). "Distributed Data Mining Bibliography". In:

Bishop, Chris M (1995). "Training with noise is equivalent to Tikhonov regularization". In: *Neural computation* 7.1, pp. 108–116.

Blott, Michaela et al. (2018). "FINN-R: An end-to-end deep-learning framework for fast exploration of quantized neural networks". In: *ACM Transactions on Reconfigurable Technology and Systems (TRETS)* 11.3, pp. 1–23.

Bottou, Léon (2008). *Stochastic gradient SVMs*. URL: http://leon.bottou.org/projects/sgd.

Boughorbel, Sabri, J-P Tarel, and Nozha Boujemaa (2005). "Conditionally positive definite kernels for SVM based image recognition". In: *2005 IEEE International Conference on Multimedia and Expo*. IEEE, pp. 113–116.

Boyd, Stephen et al. (2011). "Distributed optimization and statistical learning via the alternating direction method of multipliers". In: *Foundations and Trends in Machine Learning* 3.1, pp. 1–122.

Cadambi, Srihari et al. (2009). "A massively parallel FPGA-based coprocessor for support vector machines". In: *Field Programmable Custom Computing Machines, 2009. FCCM'09. 17th IEEE Symposium on*. IEEE, pp. 115–122.

Caruana, Godwin, Maozhen Li, and Yang Liu (2013). "An ontology enhanced parallel SVM for scalable spam filter training". In: *Neurocomputing* 108, pp. 45–57. ISSN: 09252312. DOI: 10.1016/j.neucom.2012.12.001.

Caruana, Godwin, Maozhen Li, and Man Qi (2011). "A MapReduce based parallel SVM for large scale spam filtering". In: *2011 Eighth International Conference on Fuzzy Systems and Knowledge Discovery (FSKD)* 4, pp. 2659–2662. DOI: 10.1109/FSKD.2011.6020074.

Castaldo, A.M. (2007). *Error analysis of various forms of floating point dot products*. ProQuest.

Cauchy, Augustin (1847). "Méthode générale pour la résolution des systemes d'équations simultanées". In: *Comp. Rend. Sci. Paris* 25.1847, pp. 536–538.

Chang, Chih-Chung and Chih-Jen Lin (2011). "LIBSVM: a library for support vector machines". In: *ACM Transactions on Intelligent Systems and Technology (TIST)* 2.3, p. 27.

Chang, Edward Y (2011). "PSVM: Parallelizing support vector machines on distributed computers". In: *Foundations of Large-Scale Multimedia Information Management and Retrieval*. Springer, pp. 213–230.

Chellapilla, Kumar, Sidd Puri, and Patrice Simard (2006). "High performance convolutional neural networks for document processing". In: *Proceedings of the International Workshop on Frontiers in Handwriting Recognition*. Suvisoft.

Chen, Badong, Nanning Zheng, and Jose C Principe (2013). "Survival kernel with application to kernel adaptive filtering". In: *Proceedings of the International Joint Conference on Neural Networks*. IEEE, pp. 1–6.

Chen, Badong et al. (2012). "Quantized kernel least mean square algorithm". In: *Neural Networks and Learning Systems, IEEE Transactions on* 23.1, pp. 22–32.

Chen, Yu-Hsin, Joel Emer, and Vivienne Sze (2016). "Eyeriss: A Spatial Architecture for Energy-Efficient Dataflow for Convolutional Neural Networks". In: *Proceedings of the ACM/IEEE Annual International Symposium on Computer Architecture*. IEEE.

Claesen, Marc and Bart De Moor (2015). "Hyperparameter Search in Machine Learning". In: *MIC 2015, The XI Metaheuristics International Conference*.

Cortes, Corinna and Vladimir Vapnik (1995). "Support-Vector Networks". In: *Machine learning* 20.3, pp. 273–297.

Courbariaux, Matthieu and Yoshua Bengio (2016). "Binarized Neural Networks: Training Deep Neural Networks with Weights and Activations Constrained to +1 or -1". In: *arXiv Computer Research Repository* abs/1602.02830. URL: http://arxiv.org/abs/1602.02830.

Courbariaux, Matthieu, Yoshua Bengio, and Jean-Pierre David (2015). "Binaryconnect: Training deep neural networks with binary weights during propagations". In: *Advances in neural information processing systems*, pp. 3123–3131.

Cox, David and Nicolas Pinto (2011). "Beyond simple features: A large-scale feature search approach to unconstrained face recognition". In: *Automatic Face & Gesture Recognition and Workshops (FG 2011), 2011 IEEE International Conference on*. IEEE, pp. 8–15.

Daumé III, Hal (2004). *From zero to reproducing kernel Hilbert spaces in twelve pages or less*. URL: http://legacydirs.umiacs.umd.edu/~hal/docs/ (visited on 01/12/2020).

De Figueiredo, Luiz Henrique and Jorge Stolfi (2004). “Affine arithmetic: concepts and applications”. In: *Numerical Algorithms* 37.1-4, pp. 147–158.

De Kruif, Bas J and Theo JA De Vries (2003). “Pruning error minimization in least squares support vector machines”. In: *Neural Networks, IEEE Transactions on* 14.3, pp. 696–702.

Dean, Jeffrey and Sanjay Ghemawat (2008). “MapReduce: Simplified Data Processing on Large Clusters”. In: *Commun. ACM* 51.1, pp. 107–113. ISSN: 0001-0782. DOI: 10.1145/1327452.1327492.

Detrey, J. and F. de Dinechin (2005). “A parameterized floating-point exponential function for FPGAs”. In: *Field-Programmable Technology, 2005. Proceedings. 2005 IEEE International Conference on*, pp. 27–34. DOI: 10.1109/FPT.2005.1568520.

Digilent. *PYNQ-Z1 Reference Manual*. URL: https://reference.digilentinc.com/reference/programmable-logic/pynq-z1/reference-manual (visited on 02/25/2020).

Ding, Feng (2013). “Decomposition based fast least squares algorithm for output error systems”. In: *Signal Processing* 93.5, pp. 1235–1242.

Ding, Feng et al. (2018). “Iterative parameter identification for pseudo-linear systems with ARMA noise using the filtering technique”. In: *IET Control Theory & Applications* 12.7, pp. 892–899.

Do, Thanh-Nghi, Van-Hoa Nguyen, and François Poulet (2008). “Speed up SVM algorithm for massive classification tasks”. In: *Advanced Data Mining and Applications*. Springer, pp. 147–157.

Dolbeau, Romain (2015). *Theoretical Peak FLOPS per instruction set on modern Intel CPUs*. URL: http://www.dolbeau.name/dolbeau/publications/peak.pdf.

Domke, Justin (2012a). *Statistical Machine Learning: Kernel Methods and SVMs*. URL: http://phd.gccis.rit.edu/justindomke/courses/SML/ (visited on 06/07/2019).

— (2012b). *Statistical Machine Learning: Overfitting, Model Selection, Cross Validation, Bias-Variance*. URL: http://phd.gccis.rit.edu/justindomke/courses/SML/ (visited on 06/11/2019).

Douglas, Scott C, Quanhong Zhu, and Kent F Smith (1998). “A pipelined LMS adaptive FIR filter architecture without adaptation delay”. In: *Signal Processing, IEEE Transactions on* 46.3, pp. 775–779.

Engel, Yaakov, Shie Mannor, and Ron Meir (2004). “The kernel recursive least-squares algorithm”. In: *Signal Processing, IEEE Transactions on* 52.8, pp. 2275–2285.

Esser, Steven K et al. (2016). “Convolutional Networks for Fast, Energy-Efficient Neuromorphic Computing”. In: *arXiv Computer Research Repository* abs/1603.08270.

Estrin, Gerald (1960). “Organization of computer systems: the fixed plus variable structure computer”. In: *Papers presented at the May 3-5, 1960, western joint IRE-AIEE-ACM computer conference*. ACM, pp. 33–40.

Fang, Claire Fang, Tsuhan Chen, and Rob A Rutenbar (2003). “Floating-point error analysis based on affine arithmetic”. In: *Acoustics, Speech, and Signal Processing, 2003. Proceedings.(ICASSP’03). 2003 IEEE International Conference on*. Vol. 2. IEEE, pp. II–561.

Farabet, Clément et al. (2009). “CNP: An FPGA-based processor for convolutional networks”. In: *Proceedings of the International Conference on Field Programmable Logic and Applications*. IEEE, pp. 32–37.

Forero, Pedro A, Alfonso Cano, and Georgios B Giannakis (2010). “Consensus-based distributed support vector machines”. In: *The Journal of Machine Learning Research* 11, pp. 1663–1707.

Fox, Sean, David Boland, and Philip Leong (2018). “FPGA Fastfood - A High Speed Systolic Implementation of a Large Scale Online Kernel Method”. In: *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. FPGA ’18. Monterey, CALIFORNIA, USA: ACM, pp. 279–284. ISBN: 978-1-4503-5614-5. DOI: `10.1145/3174243.3174271`.

Fox, Sean et al. (2016). “Random projections for scaling machine learning on FPGAs”. In: *Proc. International Conference on Field Programmable Technology (FPT)*. Xi’an, pp. 85–92. DOI: `10.1109/FPT.2016.7929193`.

Fraser, Nicholas J and Philip HW Leong (2020). “Kernel Normalised Least Mean Squares with Delayed Model Adaptation”. In: *ACM Transactions on Reconfigurable Technology and Systems (TRETS)* 13.2, pp. 1–30.

Fraser, Nicholas J et al. (2015a). “A fully pipelined kernel normalised least mean squares processor for accelerated parameter optimisation”. In: *Field Programmable Logic and Applications (FPL), 2015 25th International Conference on*. IEEE, pp. 1–6.

Fraser, Nicholas J et al. (2015b). “Distributed kernel learning using kernel recursive least squares”. In: *2015 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. IEEE, pp. 5500–5504.

Fraser, Nicholas J. et al. (2017a). “FPGA Implementations of Kernel Normalised Least Mean Squares Processors”. In: *ACM Trans. Reconfigurable Technol. Syst.* 10.4, 26:1–26:20. ISSN: 1936-7406. DOI: `10.1145/3106744`.

Fraser, Nicholas J et al. (2017b). “Scaling binarized neural networks on reconfigurable logic”. In: *Proceedings of the 8th Workshop and 6th Workshop on Parallel Programming and Run-Time Management Techniques for Many-core Architectures and Design Tools and Architectures for Multicore Embedded Computing Platforms*, pp. 25–30.

Frechtling, Michael and Philip HW Leong (2015). “MCALIB: Measuring sensitivity to rounding error with Monte Carlo programming”. In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 37.2, pp. 1–25.

Golomb, S. W. (1966). “Run-length Encoding”. In: *IEEE Transactions on Information Theory* 12, pp. 399–401.

Gorodnitsky, Irina F and Bhaskar D Rao (1997). “Sparse signal reconstruction from limited data using FOCUSS: A re-weighted minimum norm algorithm”. In: *IEEE Transactions on signal processing* 45.3, pp. 600–616.

Goubault, Eric (2001). “Static analyses of floating-point operations”. In: *Static Analysis*. Springer, pp. 234–259.

Graf, Hans P et al. (2004). "Parallel support vector machines: The cascade SVM". In: *Advances in neural information processing systems*, pp. 521–528.

Guyon, Isabelle et al. (2004). "Result analysis of the NIPS 2003 feature selection challenge". In: *Advances in Neural Information Processing Systems*, pp. 545–552.

Hammer, Barbara and Kai Gersmann (2003). "A note on the universal approximation capability of support vector machines". In: *Neural Processing Letters* 17.1, pp. 43–53.

Han, Song, Huizi Mao, and William J Dally (2015). "Deep Compression: Compressing Deep Neural Network with Pruning, Trained Quantization and Huffman coding". In: *arXiv Computer Research Repository* abs/1510.00149.

Hans, Mat and Ronald W Schafer (2001). "Lossless compression of digital audio". In: *Signal Processing Magazine, IEEE* 18.4, pp. 21–32.

Haykin, Simon S (2005). *Adaptive filter theory*. Pearson Education India.

He, Kaiming et al. (2017). "Mask R-CNN". In: *Proceedings of the IEEE international conference on computer vision*, pp. 2961–2969.

Hickey, Timothy, Qun Ju, and Maarten H Van Emden (2001). "Interval arithmetic: From principles to implementation". In: *Journal of the ACM (JACM)* 48.5, pp. 1038–1068.

Higham, N.J. (1996). *Accuracy and stability of numerical algorithms*. 48. Siam. ISBN: 9780898715217.

Hsu, Daniel et al. (2011). "Parallel online learning". In: *CoRR, abs/1103.4204* 3.

Hubara, Itay et al. (2016). "Quantized Neural Networks: Training Neural Networks with Low Precision Weights and Activations". In: *arXiv Computer Research Repository* abs/1609.07061.

Iandola, Forrest N et al. (2015). "FireCaffe: near-linear acceleration of deep neural network training on compute clusters". In: *arXiv Computer Research Repository* abs/1511.00175.

Iandola, Forrest N et al. (2016). "SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and <1MB model size". In: *arXiv Computer Research Repository* abs/1602.07630.

Intel. *Intel Core i7-4500U Processor*. URL: https://ark.intel.com/content/www/us/en/ark/products/75460/intel-core-i7-4500u-processor-4m-cache-up-to-3-00-ghz.html (visited on 02/25/2020).

— *Intel Xeon Processor E5-2670*. URL: https://ark.intel.com/content/www/us/en/ark/products/64595/intel-xeon-processor-e5-2670-20m-cache-2-60-ghz-8-00-gt-s-intel-qpi.html (visited on 02/25/2020).

Ioffe, Sergey and Christian Szegedy (2015). "Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift". In: *Proceedings of the International Conference on Machine Learning*. Lille, France, pp. 448–456. URL: http://jmlr.org/proceedings/papers/v37/ioffe15.html.

Jacobsen, Matthew, Yoav Freund, and Ryan Kastner (2012). “RIFFA: A Reusable Integration Framework for FPGA Accelerators.” In: *Field-Programmable Custom Computing Machines (FCCM), 2012 IEEE 20th Annual International Symposium on*. IEEE, pp. 216–219. ISBN: 978-0-7695-4699-5. URL: `http://dblp.uni-trier.de/db/conf/fccm/fccm2012.html\#JacobsenFK12`.

Jacobsen, Matthew et al. (2015). “RIFFA 2.1: A Reusable Integration Framework for FPGA Accelerators”. In: 8.4, pp. 1–23.

Jamro, E., K. Wiatr, and M. Wielgosz (2007). “FPGA Implementation of 64-Bit Exponential Function for HPC”. In: *Field Programmable Logic and Applications, 2007. FPL 2007. International Conference on*, pp. 718–721. DOI: `10.1109/FPL.2007.4380753`.

Kahan, William (1996). “The improbability of probabilistic error analyses for numerical computations”. In: *UCB Statistics Colloquium, evans hall edition*.

Kalamkar, Dhiraj et al. (2019). “A study of bfloat16 for deep learning training”. In: *arXiv Computer Research Repository* abs/1905.12322.

Karpathy, Andrej. *CS231n: Convolutional Neural Networks for Visual Recognition*. URL: `http://cs231n.github.io/convolutional-networks/`.

Kim, Minje and Paris Smaragdis (2016). “Bitwise Neural Networks”. In: *arXiv Computer Research Repository* abs/1601.06071. URL: `http://arxiv.org/abs/1601.06071`.

Kinoshita, Keisuke, Tomohiro Nakatani, and Masato Miyoshi (2006). “Spectral subtraction steered by multi-step forward linear prediction for single channel speech dereverberation”. In: *Acoustics, Speech and Signal Processing, 2006. ICASSP 2006 Proceedings. 2006 IEEE International Conference on*. Vol. 1. IEEE, pp. I–I.

Kivinen, Jyrki, Alexander J Smola, and Robert C Williamson (2004). “Online learning with kernels”. In: *IEEE transactions on signal processing* 52.8, pp. 2165–2176.

Koza, John R et al. (1996). “Automated design of both the topology and sizing of analog electrical circuits using genetic programming”. In: *Artificial Intelligence in Design96*. Springer, pp. 151–170.

Krizhevsky, Alex, Ilya Sutskever, and Geoffrey E Hinton (2012). “Imagenet classification with deep convolutional neural networks”. In: *Proceedings of the Advances in Neural Information Processing Systems*, pp. 1097–1105.

Lawrence, Neil, Matthias Seeger, and Ralf Herbrich (2003). “Fast sparse Gaussian process methods: The informative vector machine”. In: *Proceedings of the 16th Annual Conference on Neural Information Processing Systems*. EPFL-CONF-161319, pp. 609–616.

Lawson, Chuck L. et al. (1979). “Basic Linear Algebra Subprograms for Fortran Usage”. In: *ACM Transactions on Mathematical Software* 5.3, pp. 308–323. ISSN: 0098-3500. DOI: `10.1145/355841.355847`.

Le, Quoc, Tamás Sarlós, and Alex Smola (2013). “Fastfood-approximating kernel expansions in loglinear time”. In: *Proceedings of the International Conference on Machine Learning*. Vol. 85, pp. 244–252.

LeCun, Yann et al. (1998). “Gradient-based learning applied to document recognition”. In: *Proceedings of the of the IEEE* 86.11, pp. 2278–2324.

Lewis, David D et al. (2004). "Rcv1: A new benchmark collection for text categorization research". In: *The Journal of Machine Learning Research* 5, pp. 361–397.

Lin, Mingjie, Ilia Lebedev, and John Wawrzynek (2010). "High-throughput Bayesian computing machine with reconfigurable hardware". In: *Proceedings of the 18th annual ACM/SIGDA international symposium on Field programmable gate arrays*. FPGA '10. Monterey, California, USA: ACM, pp. 73–82. ISBN: 978-1-60558-911-4.

Lindstrom, F., M. Dahl, and L. Claesson (2003). "A finite precision LMS algorithm for increased quantization robustness". In: *Circuits and Systems, 2003. ISCAS'03. Proceedings of the 2003 International Symposium on*. Vol. 4. IEEE, pp. IV–365.

Liu, Weifeng, Il Park, and José C Príncipe (2009). "An information theoretic approach of designing sparse kernel adaptive filters". In: *Neural Networks, IEEE Transactions on* 20.12, pp. 1950–1961.

Liu, Weifeng, Puskal P Pokharel, and Jose C Principe (2008). "The kernel least-mean-square algorithm". In: *IEEE Transactions on Signal Processing* 56.2, pp. 543–554.

Liu, Weifeng, José C Príncipe, and Simon Haykin (2011). *Kernel Adaptive Filtering: A Comprehensive Introduction*. Vol. 57. John Wiley & Sons.

Lodhi, Huma et al. (2002). "Text classification using string kernels". In: *Journal of Machine Learning Research* 2.Feb, pp. 419–444.

Long, Guoz-hu, Fuyun Ling, and John G Proakis (1989). "The LMS algorithm with delayed coefficient adaptation". In: *Acoustics, Speech and Signal Processing, IEEE Transactions on* 37.9, pp. 1397–1405.

Lu, Bao-Liang and Masami Ito (1999). "Task decomposition and module combination based on class relations: a modular neural network for pattern classification". In: *Neural Networks, IEEE Transactions on* 10.5, pp. 1244–1256.

Lu, Bao-Liang, Kai-An Wang, and Yi-Min Wen (2004). "Comparison of parallel and cascade methods for training support vector machines on large-scale problems". In: *Machine Learning and Cybernetics, 2004. Proceedings of 2004 International Conference on*. Vol. 5. IEEE, pp. 3056–3061.

Lu, Yumao, Vwani Roychowdhury, and Lieven Vandenberghe (2008). "Distributed parallel support vector machines in strongly connected networks". In: *Neural Networks, IEEE Transactions on* 19.7, pp. 1167–1178.

Mackey, Michael C, Leon Glass, et al. (1977). "Oscillation and chaos in physiological control systems". In: *Science* 197.4300, pp. 287–289.

Majumdar, Abhinandan et al. (2012). "A Massively Parallel, Energy Efficient Programmable Accelerator for Learning and Classification". In: *ACM Transactions on Architecture and Code Optimization (TACO)* 9.1, 6:1–6:30. ISSN: 1544-3566. DOI: 10.1145/2133382.2133388.

Micikevicius, Paulius et al. (2017). "Mixed precision training". In: *arXiv Computer Research Repository* abs/1710.03740.

Misra, Janardan and Indranil Saha (2010). "Artificial neural networks in hardware: A survey of two decades of progress". In: *Neurocomputing* 74.1–3, pp. 239–255. ISSN: 0925-2312.

Moustafa, Nour and Jill Slay (2015). "UNSW-NB15: a comprehensive data set for network intrusion detection systems (UNSW-NB15 network data set)". In: *2015 military communications and information systems conference (MilCIS)*. IEEE, pp. 1–6.

Nannen, Volker (2003). *Short Introduction to Model Selection, Kolmogorov Complexity and Minimum Description Length (MDL)*. URL: http://volker.nannen.com/work/mdl/ (visited on 06/12/2019).

Naumov, Maxim et al. (2019). "Deep learning recommendation model for personalization and recommendation systems". In: *arXiv preprint arXiv:1906.00091*.

Naylor, Patrick A and Nikolay D Gaubitch (2005). "Speech dereverberation". In: *Proc. Int. Workshop Acoust. Echo Noise Control*.

Nielsen, Søren H and Thomas Lund (2000). "0 dB FS+ Levels in Digital Mastering". In: *Audio Engineering Society Convention 109*. Audio Engineering Society.

Ovtcharov, Kalin et al. (2015). *Accelerating Deep Convolutional Neural Networks Using Specialized Hardware*. URL: https://www.microsoft.com/en-us/research/publication/accelerating-deep-convolutional-neural-networks-using-specialized-hardware/.

Pakarinen, Jyri and David T Yeh (2009). "A review of digital techniques for modeling vacuum-tube guitar amplifiers". In: *Computer Music Journal* 33.2, pp. 85–100.

Pang, Yeyong et al. (2013). "A Low Latency Kernel Recursive Least Squares Processor using FPGA Technology". In: *FPT*, pp. 144–151.

Pang, Yeyong et al. (2016). "A Microcoded Kernel Recursive Least Squares Processor Using FPGA Technology". In: *ACM Transactions on Reconfigurable Technology and Systems* 10.1, 5:1–5:22. ISSN: 1936-7406. DOI: 10.1145/2950061. URL: http://dl.acm.org/authorize?N25942.

Papadonikolakis, M. and C. Bouganis (2008). "A scalable FPGA architecture for non-linear SVM training". In: *ICECE Technology, 2008. FPT 2008. International Conference on*, pp. 337–340. DOI: 10.1109/FPT.2008.4762412.

Parker, Douglass Stott (1997). *Monte Carlo Arithmetic: exploiting randomness in floating-point arithmetic*. Computer Science Department, University of California.

Pinto, Nicolas et al. (2009). "A High-Throughput Screening Approach to Discovering Good Forms of Biologically Inspired Visual Representation". In: *PLOS Computational Biology* 5.11, pp. 1–12. DOI: 10.1371/journal.pcbi.1000579.

Platt, John et al. (1998). "Sequential minimal optimization: A fast algorithm for training support vector machines". In:

Pokharel, Puskal P, Weifeng Liu, and Jose C Principe (2007). "Kernel LMS". In: *2007 IEEE International Conference on Acoustics, Speech and Signal Processing-ICASSP'07*. Vol. 3. IEEE, pp. III–1421.

Poltmann, Rainer D (1995). "Conversion of the delayed LMS algorithm into the LMS algorithm". In: *Signal Processing Letters, IEEE* 2.12, p. 223.

Pottathuparambil, Robin and Ron Sass (2009). "A Parallel/Vectorized Double-precision Exponential Core to Accelerate Computational Science Applications". In: *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays*.

FPGA ’09. Monterey, California, USA: ACM, pp. 285–285. ISBN: 978-1-60558-410-2. DOI: 10.1145/1508128.1508198.

Rasmussen, Carl E. and Christoper K. I. Williams (2006). *Gaussian Processes for Machine Learning*. Cambridge, MA, USA: MIT Press.

Rastegari, Mohammad et al. (2016). “XNOR-Net: ImageNet Classification Using Binary Convolutional Neural Networks”. In: *Proceedings of the European Conference on Computer Vision*.

Redmon, Joseph and Ali Farhadi (2017). “YOLO9000: better, faster, stronger”. In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 7263–7271.

Ren, Xiaowei et al. (2014). “Hardware Implementation of KLMS Algorithm using FPGA”. In: *Proceedings of the International Joint Conference on Neural Networks*. IEEE, pp. 2276–2281.

Rice, R. F. (1979). “Some practical universal noiseless coding techniques”. In: *Jet Propulsion Laboratory, JPL Publication*.

Richard, Cédric, José Carlos M Bermudez, and Paul Honeine (2009). “Online prediction of time series data with kernels”. In: *Signal Processing, IEEE Transactions on* 57.3, pp. 1058–1067.

Robbins, Herbert and Sutton Monro (1951). “A stochastic approximation method”. In: *The annals of mathematical statistics*, pp. 400–407.

Samuel, Arthur L (1959). “Some studies in machine learning using the game of checkers”. In: *IBM Journal of research and development* 3, pp. 210–229.

Sato, Mitsuhisa (2002). “OpenMP: Parallel Programming API for Shared Memory Multiprocessors and On-chip Multiprocessors”. In: *Proceedings of the International Symposium on System Synthesis*, pp. 109–111.

Scholkopf, Bernhard and Alexander J. Smola (2001). *Learning with Kernels: Support Vector Machines, Regularization, Optimization, and Beyond*. Cambridge, MA, USA: MIT Press. ISBN: 0262194759.

Sebald, Daniel J and James A Bucklew (2000). “Support vector machine techniques for nonlinear equalization”. In: *IEEE Transactions on Signal Processing* 48.11, pp. 3217–3226.

Seeger, Matthias (2000). “Relationships between Gaussian processes, support vector machines and smoothing splines”. In: *Machine Learning*.

Shashua, Amnon (2009). “Introduction to machine learning: Class notes 67577”. In: *arXiv preprint arXiv:0904.3664*.

Shawe-Taylor, John and Nello Cristianini (2004). *Kernel methods for pattern analysis*. Cambridge university press.

Silva, Anthony Mihirana de et al. (2013). “A Hybrid Feature Selection and Generation Algorithm for Electricity Load Prediction using Grammatical Evolution”. In: *IEEE 12th International Conference on Machine Learning and Applications ICMLA 2013, special session on Machine Learning in Energy Applications*, pp. 211–217. DOI: 10.1109/ICMLA.2013.125.

Silva, Anthony Mihirana de et al. (2016). "Forecasting Financial Time-Series with Grammar Guided Feature Generation". In: *Computational Intelligence*. ISSN: 1467-8640. DOI: 10.1111/coin.12083.

Simon, Phil (2013). *Too Big to Ignore: The Business Case for Big Data*. Wiley. com.

Simonyan, Karen and Andrew Zisserman (2014). "Very deep convolutional networks for large-scale image recognition". In: *arXiv Computer Research Repository* abs/1409.1556.

Souza, César (2010). *Kernel Functions for Machine Learning Applications*. URL: http://crsouza.com/2010/03/17/kernel-functions-for-machine-learning-applications/ (visited on 06/06/2019).

Sung, Wonyong, Sungho Shin, and Kyuyeon Hwang (2015). "Resiliency of Deep Neural Networks under Quantization". In: *arXiv Computer Research Repository* abs/1511.06488.

Suzuki, Taiji (2013). "Dual averaging and proximal gradient descent for online alternating direction multiplier method". In: *Proceedings of the International Conference on Machine Learning*, pp. 392–400.

Szalay, Alexander and Jim Gray (2006). "2020 Computing: Science in an exponential world". In: *Nature* 440.7083, pp. 413–414.

Szegedy, Christian et al. (2015). "Going deeper with convolutions". In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 1–9.

Tawfik, Sherif A (2005). *Mini-max Approximation and Remez Algorithm*.

Tay, Francis EH and Lijuan Cao (2001). "Application of support vector machines in financial time series forecasting". In: *Omega* 29.4, pp. 309–317.

Taylor, Brook (1715). *Methodus Incrementorum Directa et Inversa*. London: Gulielmi Innys.

TechPowerUp. *NVIDIA GRID K520*. URL: https://www.techpowerup.com/gpu-specs/grid-k520.c2312 (visited on 02/25/2020).

Theano Development Team (2016). "Theano: A Python framework for fast computation of mathematical expressions". In: *arXiv Computer Research Repository* abs/1605.02688.

Tikhonov, Andrey Nikolayevich (1943). "On the stability of inverse problems". In: *Dokl. Akad. Nauk SSSR*. Vol. 39, pp. 195–198.

Tridgell, Stephen et al. (2015). "Braiding: a Scheme for Resolving Hazards in Kernel Adaptive Filters". In: *Proc. International Conference on Field Programmable Technology (FPT)*. Queenstown, pp. 136–143. DOI: 10.1109/FPT.2015.7393140.

Umuroglu, Yaman et al. (2017). "FINN: A Framework for Fast, Scalable Binarized Neural Network Inference". In: *Proceedings of the ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*.

Van Vaerenbergh, S., J. Via, and I. Santamaria (2006). "A Sliding-Window Kernel RLS Algorithm and Its Application to Nonlinear Channel Identification". In: *Acoustics, Speech and Signal Processing, 2006. ICASSP 2006 Proceedings. 2006 IEEE International Conference on*. Vol. 5, pp. V–V. DOI: 10.1109/ICASSP.2006.1661394.

Van Vaerenbergh, Steven (2012). *Kernel Methods Toolbox KAFBOX: a Matlab benchmarking toolbox for kernel adaptive filtering*. Grupo de Tratamiento Avanzado de Señal, Departamento de Ingeniería de Comunicaciones, Universidad de Cantabria, Spain. Software available at `http://sourceforge.net/p/kafbox`.

Van Vaerenbergh, Steven, Miguel Lázaro-Gredilla, and Ignacio Santamaría (2012). "Kernel recursive least-squares tracker for time-varying regression". In: *Neural Networks and Learning Systems, IEEE Transactions on* 23.8, pp. 1313–1326.

Van Vaerenbergh, Steven and Ignacio Santamarıa (2013). "A comparative study of kernel adaptive filtering algorithms". In: *Digital Signal Processing and Signal Processing Education Meeting (DSP/SPE), 2013 IEEE*. IEEE, pp. 181–186.

Van Vaerenbergh, Steven et al. (2010). "Fixed-budget kernel recursive least-squares". In: *Acoustics Speech and Signal Processing (ICASSP), 2010 IEEE International Conference on*. IEEE, pp. 1882–1885.

Vapnik, Vladimir (2000). *The nature of statistical learning theory*. springer.

Venieris, Stylianos I and Christos-Savvas Bouganis (2016). "fpgaConvNet: A Framework for Mapping Convolutional Neural Networks on FPGAs". In: *Proceedings of the IEEE Annual International Symposium on Field-Programmable Custom Computing Machines*. IEEE, pp. 40–47.

Vignes, J (1996). "A stochastic approach to the analysis of round-off error propagation. A survey of the CESTAC method". In: *Proc. 2nd Real Numbers and Computers conference*, pp. 233–251.

Vishwanathan, S Vichy N et al. (2010). "Graph kernels". In: *Journal of Machine Learning Research* 11.Apr, pp. 1201–1242.

Volterra, Vito (1887). *Sopra le funzioni che dipendono da altre funzioni*. Tip. della R. Accademia dei Lincei.

Wang, Qian et al. (2013). "AUGEM: automatically generate high performance dense linear algebra kernels on x86 CPUs". In: *High Performance Computing, Networking, Storage and Analysis (SC), 2013 International Conference for*. IEEE, pp. 1–12.

Whaley, R. Clint and Antoine Petitet (2005). "Minimizing development and maintenance costs in supporting persistently optimized BLAS". In: *Software: Practice and Experience* 35.2, pp. 101–121.

Widrow, B. and M.E. Jr. Hoff (1960). "Adaptive Switching Circuits". In: *IRE WESCON Convention Record*, pp. 96–104.

Wielgosz, Maciej, Ernest Jamro, and Kazimierz Wiatr (2008). "Highly Efficient Structure of 64-Bit Exponential Function Implemented in FPGAs". In: *Reconfigurable Computing: Architectures, Tools and Applications: 4th International Workshop, ARC 2008, London, UK, March 26-28, 2008. Proceedings*. Ed. by Roger Woods et al. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 274–279.

Wilkinson, James H (1971). "Modern error analysis". In: *SIAM review* 13.4, pp. 548–568.

Wilkinson, James H. (1994). *Rounding Errors in Algebraic Processes*. Dover Publications, Incorporated. ISBN: 0486679993.

Williams, Samuel, Andrew Waterman, and David Patterson (2009). "Roofline: an insightful visual performance model for multicore architectures". In: *Communications of the ACM* 52.4, pp. 65–76.

Wolf, Helmut (1978). "The Helmert block method, its origin and development". In: *Proceedings of the Second International Symposium on Problems Related to the Redefinition of North American Geodetic Networks*, pp. 319–326.

Woodbury, Max A (1950). "Inverting modified matrices". In: *Memorandum report* 42.106, p. 336.

Xianyi, Zhang, Wang Qian, and Zaheer Chothia (2014). "Openblas". In: *URL: http://xianyi. github. io/OpenBLAS*.

Xilinx (2016). "Vivado Design Suite User Guide: High-Level Synthesis". In: *White Paper*.

Xilinx (2018a). *All Programmable 7 Series Product Selection Guide*. URL: `https://www.xilinx.com/support/documentation/selection-guides/7-series-product-selection-guide.pdf` (visited on 02/25/2020).

— (2018b). *Zynq-7000 SoC DC and AC Switching Characteristics*. URL: `https://www.xilinx.com/support/documentation/data_sheets/ds187-XC7Z010-XC7Z020-Data-Sheet.pdf` (visited on 02/25/2020).

— (2018c). *Zynq UltraScale MPSoC Product Tables and Product Selection Guide*. URL: `https://www.xilinx.com/support/documentation/selection-guides/zynq-ultrascale-plus-product-selection-guide.pdf` (visited on 02/25/2020).

— (2019a). *Alveo U200 and U250 Data Center Accelerator Cards Data Sheet*. URL: `https://www.xilinx.com/support/documentation/data_sheets/ds962-u200-u250.pdf` (visited on 02/25/2020).

— (2019b). *UltraScale FPGAs Product Tables and Product Selection Guide*. URL: `https://www.xilinx.com/support/documentation/selection-guides/ultrascale-plus-fpga-product-selection-guide.pdf` (visited on 02/25/2020).

— (2019c). *VC707 Evaluation Board for the Virtex-7 FPGA User Guide*. URL: `https://www.xilinx.com/support/documentation/boards_and_kits/vc707/ug885_VC707_Eval_Bd.pdf` (visited on 02/25/2020).

— (2019d). *Virtex UltraScale+ FPGA Data Sheet: DC and AC Switching Characteristics*. URL: `https://www.xilinx.com/support/documentation/data_sheets/ds923-virtex-ultrascale-plus.pdf` (visited on 02/25/2020).

— (2019e). *Zynq-7000 SoC Product Selection Guide*. URL: `https://www.xilinx.com/support/documentation/selection-guides/zynq-7000-product-selection-guide.pdf` (visited on 02/25/2020).

— (2019f). *Zynq UltraScale+ MPSoC Data Sheet: DC and AC Switching Characteristics*. URL: `https://www.xilinx.com/support/documentation/data_sheets/ds923-virtex-ultrascale-plus.pdf` (visited on 02/25/2020).

Yang, Jing (2006). "An improved cascade SVM training algorithm with crossed feedbacks". In: *Computer and Computational Sciences, 2006. IMSCCS'06. First International Multi-Symposiums on*. Vol. 2. IEEE, pp. 735–738.

Yates, Randy (2001). "Fixed-Point Arithmetic: An introduction". In: URL: http://www.valpont.com/wp-content/uploads/jlee/08/2015/fp.pdf (visited on 07/11/2020).

Yi, Ying et al. (2005). "High speed FPGA-based implementations of delayed-LMS filters". In: *Journal of VLSI signal processing systems for signal, image and video technology* 39.1-2, pp. 113–131.

Yuan, Ming and Yi Lin (2006). "Model selection and estimation in regression with grouped variables". In: *Journal of the Royal Statistical Society: Series B (Statistical Methodology)* 68.1, pp. 49–67.

Yukawa, M. (2012). "Multikernel Adaptive Filtering". In: *Signal Processing, IEEE Transactions on* 60.9, pp. 4672–4682. ISSN: 1053-587X. DOI: 10.1109/TSP.2012.2200889.

Zhang, Chen et al. (2015). "Optimizing FPGA-based accelerator design for deep convolutional neural networks". In: *Proceedings of the ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. ACM, pp. 161–170.

Zhou, Shuchang et al. (2016). "DoReFa-Net: Training Low Bitwidth Convolutional Neural Networks with Low Bitwidth Gradients". In: *arXiv Computer Research Repository* abs/1606.06160.