Low Latency Machine Learning on FPGAs

Stephen Tridgell

B.EEng (Hons) and B.Sc (Physics)

THE UNIVERSITY OF SYDNEY

Supervisor: Philip Leong

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

School of Electrical and Information Engineering
Faculty of Engineering
The University of Sydney
Australia

27 July 2020

Abstract

In recent years, there has been an exponential rise in the quantity of data being acquired and generated. Machine learning provides a way to use and analyze this data to provide a range of insights and services. In this thesis, two popular machine learning algorithms are explored in detail and implemented in hardware to achieve high throughput and low latency.

The first algorithm discussed is the Naïve Online regularised Risk Minimization Algorithm. This is a Kernel Adaptive Filter capable of high throughput online learning. In this work, a hardware architecture known as braiding is proposed and implemented on a Field-Programmable Gate Array. The application of this braiding technique to the Naïve Online regularised Risk Minimization Algorithm results in a very high throughput and low latency design.

Neural networks have seen explosive growth in research in the recent decade. A portion of this research has been dedicated to lowering the computational cost of neural networks by using lower precision representations. The second method explored and implemented in this work is the unrolling of ternary neural networks. Ternary neural networks can have the same structure as any floating point neural network with the key difference being the weights of the network are restricted to -1,0 and 1. Under certain assumptions, this work demonstrates that these networks can be implemented very efficiently for inference by exploiting sparsity and common subexpressions. To demonstrate the effectiveness of this technique, it is applied to two different systems and two different datasets. The first is on the common benchmarking dataset CIFAR10 and the Amazon Web Services F1 platform, and the second is for real-time automatic modulation classification of radio frequency signals using the radio frequency system on chip ZCU111 development board. These implementations both demonstrate very high throughput and low latency compared with other published literature while maintaining

very high accuracy. Together this work provides techniques for real-time inference and training on parallel hardware which can be used to implement a wide range of new applications.

Acknowledgements

To my supervisor, Professor Philip H.W. Leong, who is always enthusiastic, considerate and encouraging. The shaping of this work must have required more patience than I can imagine. A sincere thank you for all the help and advice you have provided both related and unrelated to my doctorate.

To my wonderful wife, who has never waivered in her support over the long years. Thank you for putting up with all my complaints with a smile and encouraging words.

To Dr David Boland, Mr. Nicholas Fraser, Dr. Farzad Noorian and Dr. Duncan Moss who have mentored me over the years and had the patience to teach or discuss many interesting things.

To Mr. Julian Faraone, Mr. Sean Fox, Dr. Duncan Moss and Dr. Siddhartha for being very close collaborators and for providing support and a sympathetic ear for my frustrations. I was also fortunate to collaborate with Mr. Martin Hardieck, Professor Martin Kumm and Professor Peter Zipf from the University of Kassel who showed me the sights of Kassel and the secrets of CSE.

I am very grateful to Mr. Julian Faraone, Mr. Sean Fox, Dr. Duncan Moss, Mr. Seyedramin Rasoulinezhad, Dr. Siddhartha, Dr. Andrew Tridgell, Dr. Michele Tridgell and Mrs Virginia Tridgell who provided invaluable feedback in the writing of this thesis.

Finally thank you to my family and friends who have shaped me into who I am.

Author Statement

All the content of this thesis is a product of my own work. All assistance in the preparation of this thesis have been acknowledged as follows:

- Professor Philip H.W. Leong provided the research direction
- The implementation of Naïve Online regularised Risk Minimization Algorithm was assisted by Nicolas Fraser and Duncan Moss
- The application of common subexpression elimination to unrolled ternary neural networks was conceived in a discussion with Professor Martin Kumm and Martin Hardieck
- The implementation of unrolled ternary neural networks on the ZCU111 was done in collaboration with Dr Siddhartha

Stephen Tridgell - 17/12/2019

Publications

The work presented in this thesis has been published in journals and conferences.

Journal Publications

- **Stephen Tridgell**, Martin Kumm, Martin Hardieck, David Boland, Duncan Moss, Peter Zipf, Philip HW Leong - "Unrolling Ternary Neural Networks" - *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*
- Nicholas J Fraser, Junkyu Lee, Duncan JM Moss, Julian Faraone, **Stephen Tridgell**, Craig T Jin, Philip HW Leong - "FPGA Implementations of Kernel Normalised Least Mean Squares Processors" - *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*

Conference Publications

- **Stephen Tridgell**, Duncan JM Moss, Nicholas J Fraser, Philip HW Leong - "Braiding: A scheme for resolving hazards in kernel adaptive filters" - *2015 International Conference on Field Programmable Technology (FPT)*
- **Stephen Tridgell**, David Boland, Philip HW Leong and Siddartha - "Real-time Automatic Modulation Classification" - *2019 International Conference on Field Programmable Technology (FPT)*
- Nicholas J Fraser, Duncan JM Moss, JunKyu Lee, **Stephen Tridgell**, Craig T Jin, Philip HW Leong - "A fully pipelined kernel normalised least mean squares processor for accelerated parameter optimisation" - *2015 25th International Conference on Field Programmable Logic and Applications (FPL)*
- Sean Fox, **Stephen Tridgell**, Craig Jin, Philip HW Leong - "Random projections for scaling machine learning on FPGAs" - *2016 International Conference on Field-Programmable Technology (FPT)*

Contents

List of Figures

List of Tables

Abbreviations

Abbreviation	Description
ADC	Analog to Digital Converter
AES	Advanced Encryption Standard
AI	Artificial Intelligence
ALD	Approximate Linear Dependence
AM	Amplitude Modulation
AMC	Automatic Modulation Classification
APSK	Amplitude and Phase Shift Keying
ASIC	Application Specific Integrated Circuit
ASK	Amplitude Shift Keying
AUC	Area Under the Curve
AWS	Amazon Web Services
AXI	Advanced eXtensible Interface
BN	Batch Normalization
BPSK	Binary Phase Shift Keying
BRAM	Block Random Access Memory
BU-CSE	Bottom Up - Common Subexpression Elimination
CARET	Classification And REgression Training
CIFAR	Canadian Institute For Advanced Research
CLB	Combinational Logic Block
CNN	Convolutional Neural Network
Conv	Convolutional Layer
CPU	Central Processing Unit
CSE	Common Subexpression Elimination

CSV	Comma Separated Values
CRN	Cognitive Radio Network
DAC	Digital to Analog Converter
DMA	Direct Memory Access
DNN	Deep Neural Network
DPDK	Data Plane Development Kit
DSB	Dual SideBand
DSP	Digital Signal Processing
FC	Fully Connected
FF	Flip Flop
FFT	Fast Fourier Transform
FIFO	First In First Out
FM	Frequency Modulation
FPGA	Field-Programmable Gate Array
FPS	Frames Per Second
FPT	Field-Programmable Technology
GCC	GNU Compiler Collection
GLDC	Gate Level Delay Computing
GMSK	Gaussian Minimum Shift Keying
GNU	GNU's Not Unix
GPU	Graphical Processing Unit
HLS	High Level Synthesis
IEEE	Institute of Electrical and Electronics Engineers
IP	Intellectual Property
KAF	Kernel Adaptive Filter
KRLS	Kernel Recursive Least Squares
KLMS	Kernel Least Mean Squares
KNLMS	Kernel Normalized Least Mean Squares
LC	Logic Cell
LE	Logic Element

LMS	Least Mean Squares
LSTM	Long-Short Term Memory
LUT	Lookup Table
MAC	Multiply Accumulate
MLP	Muti-Layer Perceptron
MMAC	Mega Multiply Accumulate (10^6 MACs)
MOps	Mega Operations (10^6 Operations)
NORMA	Naive Online regularised Risk Minimization Algorithm
NP	Non-Polynomial
OLK	Online Learning with Kernels
OOK	On-Off Keying
OQPSK	Offset Quadrature Phase Shift Keying
P&R	Place and Route
PCIe	Peripheral Component Interconnect Express
PEGASOS	Primal Estimated sub-GrAdient SOlver for SVM
PSK	Phase Shift Keying
PYNQ	Python Productivity for ZYNQ
QAM	Quadrature Amplitude Modulation
QKLMS	Quantized Kernel Least Mean Squares
QPSK	Quadrature Phase Shift Keying
ReLU	Rectified Linear Unit
RF	Radio Frequency
RFSoC	Radio Frequency System on Chip
RGB	Red, Green, Blue
RPAG	Reduced Pipelined Adder Graph
RTL	Register Transfer Language
SC	Suppressed Carrier
SDR	Software Defined Radio
SELU	Scaled Exponential Linear Unit
SLR	Super Logic Region

SNR	Signal to Noise Ratio
SM	Softmax
SSB	Single SideBand
SVM	Support Vector Machine
TD-CSE	Top Down - Common Subexpression Elimination
TMAC	Tera Multiply Accumulate (10^9 MACs)
TNN	Ternary Neural Network
TOps	Tera Operations (10^9 Operations)
TPU	Tensor Processing Unit
TRETS	Reconfigurable Technology and Systems
TWN	Ternary Weight Network
UCI	University of California Irvine
VGG	Visual Geometry Group
WC	With Carrier

Chapter 1

Introduction

Machine learning is a broad area with a large variety of applications. It typically requires large amounts of computational resources to be used. This thesis explores methods to improve the performance of machine learning through the use of Field-Programmable Gate Arrays (FPGAs).

The goal of machine learning is to solve tasks that would be difficult to contain as a list of rules within traditional programming techniques. A notable example would be extracting information from images. Taking two identical photos of anything is nearly impossible. There is always a slight variation in lighting or movement that subtly changes the image beyond what a human eye can detect. If a set of rules and transformations is manually created, there is almost always a case that is not considered in enough depth that causes a failure of the system. Additionally, the development of this system by hand would be very time consuming and frustrating for complex tasks such as identifying the presence of a dog in an image.

The ability of machine learning to generate these set of rules and transformations has led to its application in a wide number of fields. The problems for which machine learning have been successfully applied are as diverse as translation [5], fraud detection [11] and object tracking [87]. The world has changed significantly due to the ability to solve these problems and enabled the implementation of services that were previously infeasible.

Machine learning attempts to generate what could be thought of as a detailed and generalized set of rules and transformations to perform a specified task called a model.

In some cases, these rules are directly interpretable such as in decision trees where there is a choice based on some feature. Alternatively, the transformations can be much more abstract, represented as a matrix of values in a neural network. In order to create this model with machine learning, a large number of examples are typically needed to capture the variety of the input. For simpler tasks, such as differentiating between red or green apples, fewer examples are needed as it is easier to find a model that capture this information. More examples are required for complex tasks to subtly differentiate between inputs.

The appearance of cats and dogs is diverse and can have similar features such that even humans can mistake one for the other. Simple rules such as recognizing pointy or floppy ears may capture a majority of cases but still fail on quite a number. Images of cats and dogs that have other characteristics enable the development of more subtle rules and edge cases for distinguishing between them, improving the performance.

The generation of the model in machine learning is also referred to as *training*. After the model has been generated to a satisfactory level, they can be used for the desired application. Using the model in this way is known as *inference*. One significant limitation of machine learning is the computational cost required for both training and inference. In addition to the purchase and replacement of expensive components, the power cost to run the computational platforms and cooling can be quite substantial [27, 77].

Finding the right balance between the cost of creating the model and the accuracy of the prediction is a trade-off to be considered. Different applications will require varying degrees of accuracy or speed in both training and inference. In cases where very high accuracy is required, larger models are used which need more computation. Where speed is needed, smaller models can be used as well as other simplifications and sacrifices to reduce the complexity of the model.

There are two aspects of the speed of computation that need to be considered. The first aspect is *throughput* or how many examples can be processed in a given time. The

other is *latency* or how long before the result for a particular set of inputs is available. These terms are can be easily interpreted with an analogy of a post office. How many letters the post office can deliver in a day is the throughput. The latency is the time to deliver a single letter after is it posted. There is typically a trade-off between these two measures of speed. Continuing with this analogy, there could be postman waiting for each letter ready to rush it to its destination. The latency in this case would be very low. However assuming the same number of postmen are employed, the number of letters they could deliver in a day is much lower. This means a significant drop in throughput for the improvement in latency. Alternatively, the post office could hold letters and wait until everyone on the street has mail before sending someone out to deliver it. This system would be able to cope with a very large volume of letters, meaning a high throughput. However, in some cases, people would have to wait a long time for the delivery if there is insufficient mail for their street. This would create unacceptable latency. The latency and throughput requirements can vary greatly depending on the application and are almost always related.

In machine learning, the performance requirements for training and inference can be different: for example, in a massive system such as the Google search. Google search has to deal with a high throughput of websites changing all the time during its training phase. The latency in this phase is relatively unimportant in comparison. However, during the inference phase when somebody wishes to query Google, how long it takes to produce results is critical. Users do not wish to wait for the result so low latency is essential. In this case, the number of active users is also significant requiring a high throughput for the inference phase.

In other applications, training time is not critical compared with inference: for example, the computation of a speech processing model deployed on a wide scale. In this case, it is important for the model to be able to understand and respond to the speech in a short enough time frame. The amount of time spent to train that model initially is not as important. Optimizing the inference of the model will give the desired results.

One interesting domain of problems where minimal training and inference time is required is known as *Online Learning*. Online learning is where the model can be updated in real-time, adapting to new unseen inputs. This is also useful in cases where the behaviour of what you are trying to predict changes frequently and looking at the past is only minimally useful. An example could be the stock market where the behavior changes frequently. Online learning approaches typically have low cost training and inference components where these steps can frequently be combined. This makes them useful for very large datasets where other techniques are infeasible.

The number of applications to which machine learning is applied is expanding rapidly. However, the types of problems can be broadly grouped into three categories. These are:

- Supervised Learning
- Unsupervised Learning
- Reinforcement Learning

Supervised learning requires a labelled dataset which means, given some example, the label is known. For example, an image of a cat is labelled 'cat'. This simplifies the training of the model as it can easily be corrected with the labelled data. The disadvantage of this approach is that collecting labelled data can be expensive and time consuming. For example, to be used to recognize hand written numbers, supervised learning requires each image of the numbers and have a label. This is label might be created by a human recording what number they think it is.

Where there is no label, the domain is known as unsupervised learning. This domain offers broad generality but is very difficult to train effectively. For the hand-written numbers example, unsupervised learning might try to group the numbers together. However as numbers like 1 and 7 can appear similar, these are likely to be grouped together. Reinforcement learning is a relatively new area. It trains by performing actions with feedback from the environment. For example, it may attempt to draw a number on a piece of paper with a robotic arm. Another system would be used to score

how well it drew the desired number providing feedback. Each of these domains have different applications. The domains can overlap as applications can have features of many domains. Machine learning models can normally be used in any of these domains with a different wrapper around how they are trained. The majority of this thesis will explore supervised learning as the purpose is to experiment with acceleration of machine learning methods rather than exploring the theory.

Training a model consumes a significant amount of resources [76]. There are a four main computational resources that are used for the evaluation of models. These are:

- Central Processing Units (CPUs)
- Graphical Processing Units (GPUs)
- Field Programmable Gate Arrays (FPGAs)
- Application-Specific Intergrated Circuits (ASICs)

Each of these resources available have different advantages and disadvantages.

CPUs are the most versatile. They function by partitioning the problem into a series of simple sequential tasks. These tasks are known as instructions which modify the state of the CPU. For applications that must be executed in sequential order, CPUs are typically the best choice. Most machine learning tasks however, have a reasonable degree of independence between tasks also known as parallelism. Despite this, nearly all real-world systems have some form of CPU to manage the resources and Input/Output (I/O).

GPUs were originally designed to accelerate transformations needed to display images, typically in gaming. These transformations can be expressed as linear algebra transformations, which while costly, do not have many dependencies inside the calculation. The approach to the initial design of GPUs was an array of simple and slower CPUs that could all be used at once. Currently, they are very widely used for machine learning applications due to the large amount of parallel computation they offer.

FPGAs are more unusual computational devices. They do not have the concept of instructions in contrast to CPUs and GPUs and instead offer programmable hardware that can represent the problem or task. Due to greater flexibility, it is typically more time consuming to create an application. However, FPGAs do have a few advantages from a computational perspective over CPUs and GPUs. They typically have lower clock frequencies than GPUs and CPUs which leads to lower power consumption. They also are not restricted to working within the fixed set of instructions of GPUs and CPUs that creates overhead in some applications. Interfacing to sensors or external devices with a FPGA typically has much lower latency than CPUs or GPUs.

ASICs offer the most flexibility and potential computation as they can implement any desired hardware. In theory, they can implement any of the other systems discussed and can surpass them by being more specialized for the application. In practice, implementing any of the other systems is exceedingly expensive and time consuming.

This thesis will explore machine learning on FPGAs and the potential offered with a hardware approach. The lower power consumption and ability to interface with sensors means it is well suited for peripheral devices for applications such as edge AI. Additionally, the reprogramability means it is more flexible to design changes than an ASIC and requires less development time.

Each machine learning method has advantages and disadvantages to be considered. Some of the most commonly used methods are:

- Kernel methods
- Neural networks
- Decision trees
- Probabilistic methods
- Genetic algorithms

Kernel methods are an approach to machine learning from a statistical perspective. They refer to a suite of algorithms that create a predictive model by choosing a set of

examples to represent the dataset. This is achieved using a mathematical tool called a kernel to measure the similarity between examples. With this, they attempt to choose a minimal set of examples that adequately represents the problem and use them to determine the predicted value for the new example.

Research into neural networks has been ongoing for many decades. Neural networks originated in trying to model the activity of neurons in the brain. They have evolved over time to a diverse suite of models and have been successfully applied to many different applications. In the past decade, the pace of research into neural networks has increased dramatically as computation became faster and cheaper through the use of GPUs.

Decision tree methods attempt to create a series of rules from the dataset. These rules are generated from the given examples by finding boundaries which split the examples. In addition to just using a single decision tree, there is a variety of ways to combine them to dramatically improve the effectiveness. One of the most popular decision tree methods is random forests [8]. Instead of using a single decision tree to create the model, a 'forest' of trees is created. Each of the trees in this forest is trained on a random portion of the dataset to give each tree a different perspective on the problem. Combining these trees dramatically improves the accuracy of the model.

Probabilistic methods, such as Bayesian networks, try to create a model based on the relationships between different variables or events. These can generate an effective representation of a system to accurately predict outcomes.

Genetic algorithms use the concept of evolution to gradually improve a model. By randomly changing something and determining if it improves the performance, the model slowly converges on the solution. Typically many models are altered in parallel and then combined to keep desirable traits.

These methods can also be combined taking aspects of two or more methods such as Kontschieder et. al. [42]. This thesis will explore two main areas within machine

learning which are kernel methods and neural networks. Decision trees, probabilistic methods and genetic algorithms will not be explored in this work.

1.1 Contributions and Aims

This thesis explores different machine learning methods and examines or proposes techniques for implementing them effectively in hardware. The work strives to improve the throughput and lower the latency of two machine learning methods in particular. These are the Naïve Online regularised Risk Minimization Algorithm (NORMA) and convolutional neural networks with ternary weights or Ternary Neural Networks (TNNs). For these two algorithms this thesis describes the following contributions:

- A novel technique for implementing NORMA on an FPGA called *braiding*
- The highest throughput and lowest latency implementation of a Kernel Adaptive Filter on an FPGA
- An unrolling technique to apply to TNNs that enables the efficient exploitation of sparsity for convolution operations
- The application of Common Subexpression Elimination (CSE) to this unrolling technique to significantly reduce the hardware cost for implementation
- An open source pip3 package **twn_generator** for easy Verilog generation of these techniques
- A high accuracy implementation on a AWS-F1 instance for the CIFAR10 dataset, achieving high throughput and low latency
- Consideration of the activations precision to design high accuracy and compact networks in hardware
- The application of this unrolling technique to radio modulation classification on the RFSoC board

1.2 Structure of the Thesis

An overview of relevant machine learning theory is provided in Chapter 2. This includes the machine learning formations of kernel methods and deep neural networks.

Chapter 3 explores the applications of hardware acceleration techniques to kernel methods. In particular this thesis proposes a hardware acceleration technique referred to as *braiding* to implement the kernel adaptive filter (KAF) algorithm known as NORMA on an FPGA with high throughput and low latency.

Chapter 4 explores hardware acceleration of neural networks. The main contribution of this chapter is a methodology to implement convolutional neural networks on an FPGA with ternary weights. It describes how to implement various hardware blocks on an FPGA for unrolled ternary neural networks.

Chapter 5 investigates the effectiveness of the methodology discussed in Chapter 4 with two different case studies. The first case study is the common benchmarking dataset, CIFAR10, which contains various images from each of 10 classes. An AWS F1 instance is used to accelerate the neural network to achieve high throughput and low latency inference. The second case study explores the domain of Automatic Modulation Classifcation (AMC) which involves classifying various radio frequency modulations. This is implemented on the RFSoC board or the ZCU111 chip which has up to 4 GHz sample rate Analog to Digital Converters (ADCs) and Digital to Analog Converters (DACs).

Chapter 6 summarizes the contributions of this thesis and discusses possible areas for future research.

Chapter 2

Background

In recent years, there has been an exponential rise in the quantity of data being acquired and generated. Attempting to extract useful information from the complex relationships present in such data has led to an increasing interest in machine learning. The motivation for using machine learning is to provide generalization to unseen examples. This is particularly useful in areas which capture complex information from the environment such as images or audio where every example will be slightly different. There have been many approaches over the years to try and achieve this generalized knowledge of a problem. Two categories of machine learning algorithms that are explored in this thesis are kernel methods and Deep Neural Networks. In this chapter, Section 2.1 explores the theory behind the large variety of kernel methods. Deep Neural Networks are discussed in Section 2.2.

In this thesis, it will be assumed that the dataset can be represented as a set of pairs, $\{x_i, y_i\}$. x_i is the input vector and y_i is the output, target or label. In general, the goal of machine learning is to create a model which can accurately predict y, given x. The dataset is often partitioned into a training portion and a test portion. This is to obtain an effective measurement of the performance of a model on the unseen data.

It is relatively easy to generate a model that perfectly predicts all the examples in the training set. In most cases however, this model performs poorly on the unseen test data. This phenomenon is known as overfitting, where the model has failed to grasp the big picture and instead has focused on the tiny differences. Figure 2.1 demonstrates the concept of overfitting. A curve is fitted to a collection of points to model the relationship between two variables. This curve perfectly predicts every example given. This is not

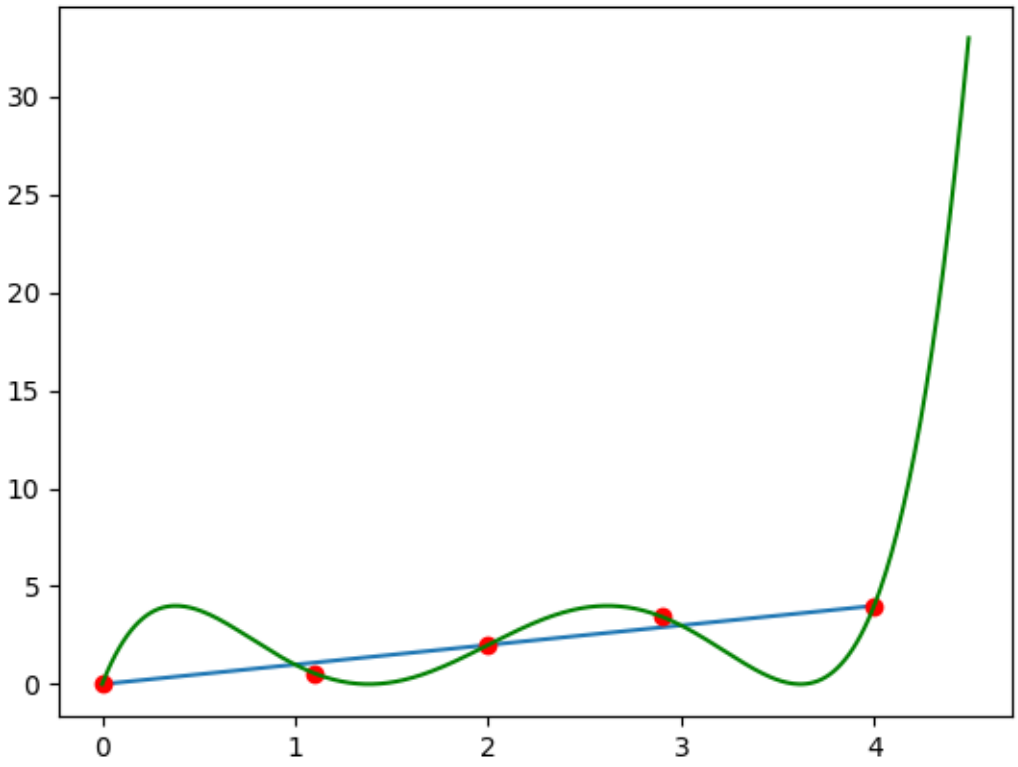


Figure 2.1: Demonstration of Overfitting

that informative however. In reality, a simpler curve provides a much clearer picture of the relationship. Occam's razor states *Entities should not be multiplied without necessity.* In this context, the green curve that fits all the examples is far more complex than the blue line in Figure 2.1. The green curve has been 'multiplied without necessity' to create a prediction that is likely to not generalize very well. Trying to find the simplest model that correctly classifies the training set is the main goal of machine learning methods.

2.1 Kernel Methods

Kernel methods are a popular class of machine learning algorithms which are capable of modelling any continuous function with arbitrary accuracy [28]. The power of kernel methods comes from the mapping of examples to a higher dimensional feature space, $\mathbb{F}$. This higher dimensional feature space allows the clearer separation or grouping of examples. Each example is mapped using the function $\phi : \mathbb{R}^m \to \mathbb{F}$. In this feature space two examples, x_i and x_j, can then be compared using the inner product. This is defined as the kernel function, $\kappa(x_i, x_j) = \phi(x_i)^T \phi(x_j)$.

Some common kernel functions include:

- the linear kernel, $\kappa(x_i, x_j) = x_i^T x_j$;
- the Gaussian kernel, $\kappa(x_i, x_j) = \mathrm{e}^{-\gamma \|x_i - x_j\|_2^2}$; and
- the polynomial kernel, $\kappa(x_i, x_j) = (x_i^T x_j + c)^b$.

where γ, b and c are parameters chosen to suit the given problem.

If the feature space is carefully chosen such as the ones above, the kernel can be computed more efficiently in a lower dimensional space while maintaining the advantages of a higher dimensional feature space. This is often referred to as the *kernel trick* [74]. To demonstrate the benefit of the kernel trick, consider the polynomial kernel $\kappa(q, r) = (q^T r)^2$ where $q, r \in \mathbb{R}^3$ or q and r are three dimensional vectors.

$$\phi(x) = (x_1^2, \sqrt{2}x_1x_2, x_2^2, \sqrt{2}x_1x_3, \sqrt{2}x_2x_3, x_3^2) \tag{2.1}$$

$$\phi(q)^T \phi(r) = q_1^2 r_1^2 + 2q_1 r_1 q_2 r_2 + q_2^2 r_2^2 + 2q_1 r_1 q_3 r_3 + 2q_2 r_2 q_3 r_3 + q_3^2 r_3^2 \tag{2.2}$$

$$= (q_1 r_1 + q_2 r_2 + q_3 r_3)^2 = (q^T r)^2 = \kappa(q, r) \tag{2.3}$$

The feature space mapping for this kernel is given in Equation 2.1. Equation 2.2 expands the inner product between q and r in feature space. This gives the same expression for the expanded kernel in Equation 2.3.

From these methods, there are two ways to compute the kernel function $\kappa(q, r) = \phi(q)^T\phi(r)$. The first method would be to take q and r separately and compute their feature space representation using Equation 2.1. In this case, this would be $1 + 2 + 1 + 2 + 2 + 1 = 9$ multiplications to compute each of $\phi(q)$ and $\phi(r)$ or 18 multiplications total. After that, the next step would be to compute the dot product between these two vectors in feature space resulting in another 6 multiplications and 5 additions. Computing the result in feature space for this polynomial kernel would then need a total of 24 multiplications and 5 additions.

If instead, the kernel trick is used to compute the result, Equation 2.3 shows taking the dot product between q and r only needs 3 multiplications and 2 additions. This result is then squared meaning another multiplication is needed, taking the total to 4 multiplications and 2 additions. This is significantly less than computing the result in feature space. This saving in computation, while still computing in feature space, is known as the kernel trick.

Different problems can be separated more effectively with different feature mappings and hence have different kernel functions. This thesis assumes the use of the Gaussian kernel due it being a universal approximator [28].

More intuitively, a kernel function can be thought of as measuring the similarity between two examples. Considering the Gaussian kernel, the first component $\|x_i - x_j\|_2^2$ is measuring the distance between two vectors x_i and x_j. Figure 2.2 illustrates the Gaussian kernel function of e^{-x^2}. If the distance between x_i and x_j is 0 ($x^2 = 0$), the *similarity* of these vectors is given a value of 1 as shown in the figure. As the distance between these vectors grows, the *similarity* decreases until it eventually will reach 0.

Kernel methods create a decision function, $y_i = f(x_i)$ using a kernel function and learned parameters. The kernel function takes two input feature vectors of length m and outputs a single real number. More formally the kernel function is defined as $\kappa : \mathbb{R}^m \times \mathbb{R}^m \to \mathbb{R}$. The learned parameters to generate the decision function include:

- a *dictionary*, $\mathcal{D}$, a subset of input vectors; and

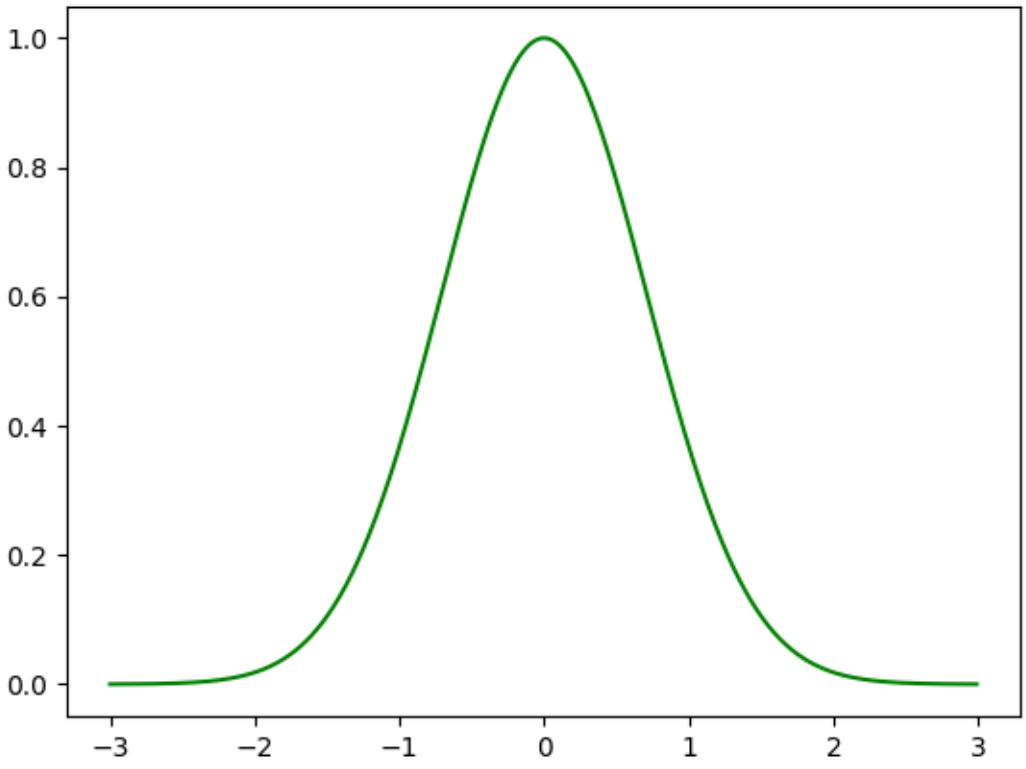


Figure 2.2: A Gaussian Function

- a vector of weights, $\boldsymbol{\alpha}$, with a scalar for each entry in the dictionary.

Using the dictionary and weights, $f(x)$, is defined as follows:

$$f(x) = \sum_{i=1}^{D} \alpha_i \kappa(x, d_i) \quad , \tag{2.4}$$

where D is the number of entries in the dictionary, α_i is the i^{th} element of $\boldsymbol{\alpha}$ and d_i is the i^{th} entry of $\mathcal{D}$. With the interpretation of the kernel function as a measure of similarity, Equation 2.4 then shows the measurement of similarity between the new example x and each example in the dictionary d_i. After these similarities are measured, they are weighted with α_i and added together. This weight ranks the importance of the examples in addition to using the sign to indicate the class (1 or -1). Based on the value of the result, the example can be classified. All kernel methods have a decision function

similar to Equation 2.4. They vary significantly in how they select the dictionary and the weights α_i.

2.1.1 Support Vector Machines

Support Vector Machines (SVMs) were derived through the application of advanced statistical techniques [74]. The main idea behind the derivation of SVMs is that, given a dataset with two classes, where is the best boundary to minimize the risk of a misclassification on unseen data. The SVM creates this boundary by selecting a subset of the training examples. Within this subset, the examples are weighted with their importance using the α_i from Equation 2.4.

There are many variations of the SVM which explore various trade-offs. They all still result in the evaluation function described in Equation 2.4 but vary in how they select the dictionary and weights. In the original derivations of the SVM, it searched for a boundary that would separate all examples from both classes known as a *hard-margin* SVM. In the real world, some of the examples might be mislabeled or have noise that causes them to appear within the cloud of examples from the other class. Rather than try to complicate the boundary by adding more complexity, a *soft-margin* recognizes noise in examples and allows for misclassifications. This is the most common variation of the SVM.

Figure 2.3 is a visualization of the soft-margin for the SVM. In this figure, the SVM boundary is trying to distinguish between red and blue examples. The dashed line indicates the boundary between classes, the green lines indicate the margins. In the figure there are examples on the wrong side of the margin meaning it is not possible to have a linear boundary separating all red and all blue points. A non-linear boundary could be created such as the black line but unseen examples such as the black cross and circle would probably be misclassified. A soft-margin creates a simple boundary that is more likely to generalize well to unseen examples.

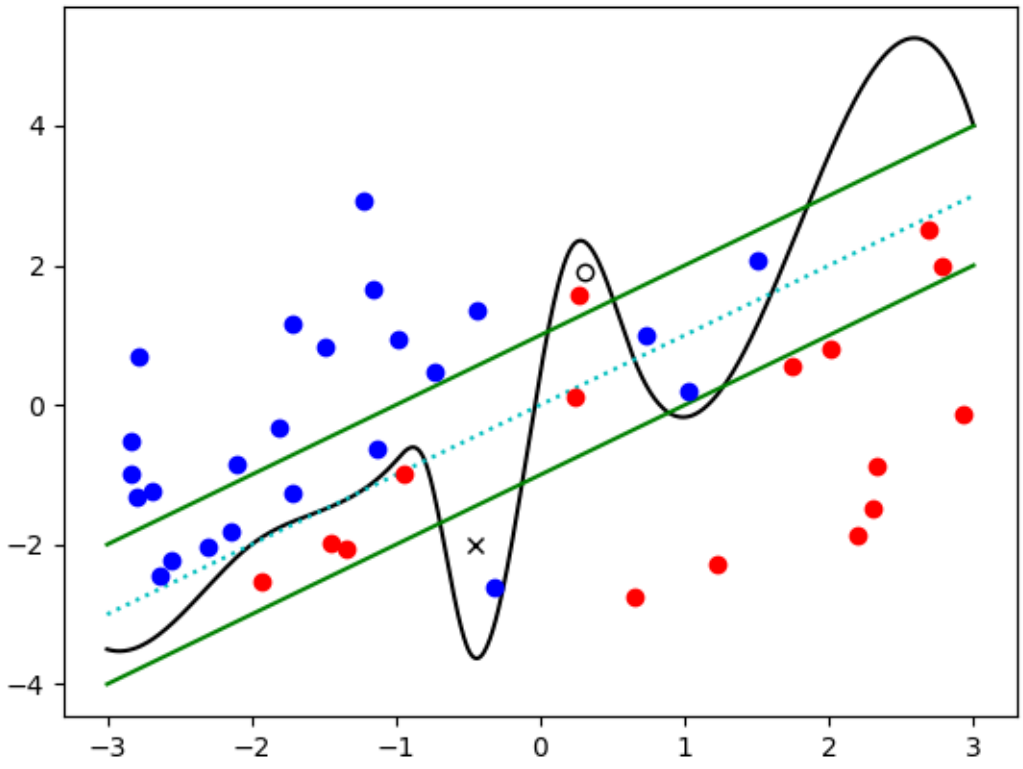


Figure 2.3: A Visualization of the soft-margin

Hinge loss is typically included with a soft-margin. The penalty term for a soft-margin with hinge loss is:

$$\max(0, 1 - y_i f(x_i)) \tag{2.5}$$

Equation 2.5 can be intuitively explained as a hinge swinging between two cases. In the first case, if the example is correctly classified the penalty is 0. If this case was not considered, the soft-margin would set examples that are correctly classified and far away from the margin to large negative values. In an optimization function, this would mean misclassifications might be ignored in favour of moving the margin further away from correctly classified examples. In the other case of hinge loss the soft-margin is implemented. As the decision moves inside the margin and away from being completely

correctly classified the penalty grows. As this term is minimized in the SVM algorithm, it pushes the model towards a lower error rate.

Only optimizing the loss term in Equation 2.5, the boundary created would still be very complex and computationally intensive. Additionally, it would likely overfit meaning the ability of the model to generalize to unseen data would be limited. Considering this in relation to the decision function in Equation 2.4, the boundary includes the weights $\boldsymbol{\alpha}$ which are non-zero for each support vector. If these weights are pushed towards zero, fewer support vectors are included leading to a simpler boundary. This is the purpose of the regularization term. For the SVM formulation the regularization term given in Equation 2.6.

$$\frac{1}{2} \|w\|_2 \tag{2.6}$$

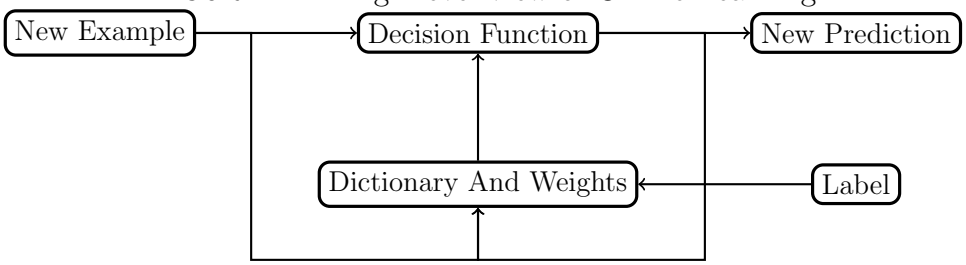
Collectively minimizing the regularization term and loss function leads the SVM formulation which is a compromise between a simple boundary while maintaining the correct classifications.

2.1.2 Kernel Adaptive Filters

If the dataset accurately represents the unseen data, SVMs work well to generate a boundary to use as a predictive model. In online and streaming applications, this is not always the case. As such, the model deduced by a machine learning algorithm must be updated as new data becomes available. Kernel Adaptive Filters (KAFs) [53] are a class of online, non-linear learning algorithms with substantially reduced computational requirements for an online setting. The reduction is achieved by finding a model update step based on the previous model and a new data example. There are many variations of KAFs but they all involve updating the dictionary, $\mathcal{D}$, and the weights for those examples, $\boldsymbol{\alpha}$.

Figure 2.4 shows a high level perspective of KAFs. A new example arrives for the KAF which is passed to the decision function. This uses the dictionary and weights to generate a prediction. The new example, a label and often some components of the

Figure 2.4: High level view of Online Learning



decision function are used to determine how the dictionary and weights are updated. After they are updated, the KAF is ready to accept a new example for prediction.

One method of updating the weights involves the adaptation of the Recursive Least Squares algorithm to a kernel space, known as Kernel Recursive Least Squares (KRLS) [17]. Another example is the Kernel Least Mean Squares (KLMS) algorithm [52]. Both of these methods involve slowly adding more terms to the dictionary and updating existing ones to converge on a good boundary. The most computationally difficult part of these algorithms is a matrix inversion update each iteration. This computation is $O(n^2)$ each step.

KAFs typically choose to add more examples as time goes on. In practice, this could grow indefinitely leading to high memory and computation requirements. To avoid this issue, the least significant examples are discarded from the dictionary after there are more than D entries. The simplest way to implement this is with a sliding window where the oldest examples are discarded if a new example is to be added.

2.1.2.1 Kernel Recursive Least Squares

The KRLS algorithm [17] functions in the same way as Figure 2.4. After receiving a new example, each entry in the current dictionary is used to compute the kernel function against it. This will be referred to as $\kappa_{t-1}(x_t)$. By multiplying this result with the weights α_{t-1}, the new prediction can be produced.

The new example, the label and $\kappa_{t-1}(x_t)$ are then passed to the update algorithm for consideration. The next step in the KRLS algorithm is to determine if the new

example should be added into the dictionary or not. KRLS leverages the approximate linear dependence (ALD) condition to check if the label of the new example can be approximated within ν^2 error. If it cannot be approximated within this error bound, the example must be added to the dictionary. Either way the weights are still updated. The most computationally complex part of this algorithm is the recursive update of the inverse kernel matrix needed for the ALD condition. This step has computational complexity of $O(D^2)$ where D is the size of the dictionary.

The dictionary and corresponding weights are built up by the algorithm one example at a time. These form the decision function that can be used online or offline. The KRLS algorithm is not considered further in this thesis, so for further details and derivation see Engel et. al. [17].

2.1.2.2 $O(n)$ Stochastic SVM approximations

The computational complexity of SVM ($O(n^3)$) [74] and KRLS ($O(n^2)$) [17] are not scalable for use in high frequency, real-time applications that also require online learning. Some example applications include channel equalisation and machine prognostics [16, 34, 66].

There are numerous proposed $O(n)$ stochastic SVM approximations. Some of the most well known ones include NORMA [39], Online Learning with Kernels (OLK) [49] and Primal Estimated sub-GrAdient SOlver for SVM (PEGASOS) [75]. All of these methods implement the *Dictionary and Weights* section of figure 2.4 through a simple constant multiple decay of the existing weights and conditionally adding a new dictionary entry with some predetermined weighting.

As these algorithms are very similar, this thesis will just explore the very widely known and original implementation of NORMA in more detail. The methodologies discussed later could also be applied to these other similar algorithms with minor modification.

2.1.2.3 Naïve Online regularised Risk Minimization Algorithm

NORMA is derived as a $O(n)$ stochastic approximation of the SVM [39]. It is an online algorithm which can be applied to classification, regression or novelty detection. NORMA is based on the concept of minimizing the instantaneous risk of predictive error by taking a step in that direction given by:

$$f_{t+1} = f_t - \eta_t \partial_f R_{inst,\lambda}[f, x_{t+1}, y_{t+1}]\Big|_{f=f_t} \tag{2.7}$$

where f_t is the function, f, at time t, η is the step size, ∂ is the partial derivative operator and $R_{inst,\lambda}$ is the instantaneous risk function. Equation 2.7 is then expressed in terms of a loss function, $l(f_t(x_t), y_t)$, to penalize misclassifications or predictions of the function given by:

$$f_{t+1} = (1 - \eta\lambda) f_t - \eta_t l'(f_t(x_{t+1}), y_t)\kappa(x_{t+1}, \cdot) \quad . \tag{2.8}$$

Depending on the application, a suitable loss function can be provided to achieve a specific goal. For example:

$$l(f(x) + b, y) = \max(0, \rho - y(f(x) + b)) - \nu\rho \tag{2.9}$$

$$l(f(x)) = \max(0, \rho - f(x)) - \nu\rho \tag{2.10}$$

$$l(f(x), y) = \max(0, |y - f(x)| - \epsilon) + \nu\epsilon \tag{2.11}$$

where η, λ, $\nu \in \mathbb{R}$ are parameters, are used for classification (2.9), novelty detection (2.10), and regression (2.11). It would be trivial to extend this work to other loss functions.

Reference [39] provides a complete derivation of Equations 2.7 and 2.8. In this thesis, the factor decaying the weights of f_t or $(1 - \eta\lambda)$ in Equation 2.8 is defined as the forgetting factor Ω. Differentiating the loss functions then substituting them into Equation 2.8

leads to the following update expressions:

$$(\alpha_i, \alpha_t, b, \rho) = \begin{cases} (\Omega\alpha_i, 0, b, \rho + \eta\nu) \text{ if } y(f(x) + b) \geq \rho \\ (\Omega\alpha_i, \eta y, b + \eta y, \rho - \eta(1 - \nu)) \text{ otherwise} \end{cases} \tag{2.12}$$

$$(\alpha_i, \alpha_t, \rho) = \begin{cases} (\Omega\alpha_i, 0, \rho + \eta\nu) \text{ if } f(x) \geq \rho \\ (\Omega\alpha_i, \eta, \rho - \eta(1 - \nu)) \text{ otherwise} \end{cases} \tag{2.13}$$

$$(\alpha_i, \alpha_t, \epsilon) = \begin{cases} (\Omega\alpha_i, 0, \epsilon - \eta\nu) \text{ if } |y - f(x)| \leq \epsilon \\ (\Omega\alpha_i, \delta\eta, \epsilon + \eta(1 - \nu)) \text{ otherwise} \end{cases} \tag{2.14}$$

where $\delta = sign(y - f(x))$, $\Omega \leq 1$ and $f(x) = \sum \alpha_i \kappa(x, d_i)$. Equations 2.12, 2.13 and 2.14 are the update expressions for classification (NORMA$_c$), novelty detection (NORMA$_n$) and regression (NORMA$_r$) respectively. The expression determining the constant value Ω with respect to other parameters varies for the application. For the classification and novelty detection loss functions in this work $\Omega = 1 - \eta$ whereas $\Omega = 1 - \lambda\eta$ for

regression. In this thesis, Ω is used for simplicity to be consistent for all applications to refer to the value that decays the weights.

Algorithm 1: Pseudocode for online training of NORMA$_c$

$\alpha = D * [0]$;
$d = D * [F * [0]]$;
$b = 0$;
$\rho = 0$;
for *x, y in Examples* **do**
  $y_{pred} = \sum \alpha[i]\kappa(x, d[i])$;
  $\alpha = \Omega\alpha$;
  if $y * (y_{pred} + b) \geq \rho$ **then**
    Ignore x;
    $\rho = \rho + \eta\nu$;
  else
    Add x to the dictionary;
    $\alpha[1 \text{ to } D] = \alpha[0 \text{ to } D-1]$;
    $d[1 \text{ to } D] = d[0 \text{ to } D-1]$;
    $b = b + \eta y$;
    $\rho = \rho - \eta(1-\nu)$;
    $\alpha[0] = \eta y$;
    $d[0] = x$;
  end
end

As discussed in Section 2.1.2, the dictionary would grow indefinitely meaning high memory and computation requirements. To manage this NORMA is used with truncation, commonly known as a sliding window, in which $\mathcal{D}$ is updated according to

$$[d_1, \cdots d_D] \rightarrow [x_t, d_1, \cdots d_{D-1}] \tag{2.15}$$

The examples retained by the function $f(x)$ are called Support Vectors for machine learning techniques such as SVMs but are more commonly referred to as the *dictionary* in the KAF literature. The example i in the dictionary is referred to d_i in this work. The corresponding weights are referred to as α_i as defined in equations 2.12, 2.13 and 2.14. One significant observation that is exploited in Chapter 3 is that after the computation of $f(x)$, the update of the weights is a scalar multiplied by a vector while rest of the update expression consists of trivial scalar expressions easily implemented in hardware. To clarify this observation the pseudocode for the NORMA algorithm is provided in figure 1. In this code, D is the size of the dictionary $\mathcal{D}$ and F is the number of features in each example.

In Chapter 3, a hardware architecture is proposed for the implementation of the well-known algorithm NORMA [39], a KAF capable of regression, classification and novelty detection. The architecture features a fully pipelined datapath, with parallel execution and conditional forwarding to overcome all data hazards whilst incurring minimal increase in the number of arithmetic operations. A technique called *braiding*, is introduced in Chapter 3 in which: (1) all partial results are computed in parallel; (2) when the information regarding which results should be included becomes available, an appropriate select signal is generated; and (3) a multiplexer selects the appropriate partial sum to add to the full sum.

2.2 Deep Neural Networks

Deep Neural Networks (DNNs) [46] are a class of machine learning algorithms that are described as a connected graph of basic compute nodes called neurons. Each layer l performs a mapping $\mathbb{R}^M \rightarrow \mathbb{R}^N$, where $\mathbf{x^l}$ is the input vector, $\mathbf{a^l}$ is the output or activation vector and b_i^l is the bias term. The computation for each neuron is the inner product between the input and its weight vector, $\mathbf{w_i^l} \in \mathbb{R}^N$, followed by an activation function f which is typically tanh, ReLU, sigmoid, etc. The operation performed to

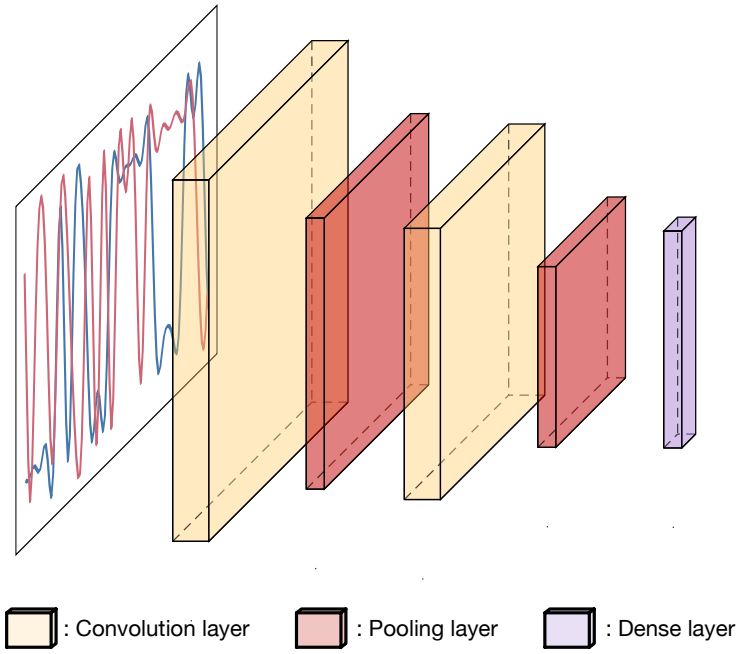


Figure 2.5: Example CNN

compute the i'th output in layer l is computed as

$$a_i^l = f(\mathbf{w_i^l x^l} + b_i^l) \tag{2.16}$$

DNNs are arranged such that the output from one layer is passed to the input of neurons in a subsequent layer. These are named dense layers. By stacking multiple dense layers together, complex nonlinear interactions can be modeled. These network topologies are named multilayer perceptrons (MLP). If inference is performed with a single input vector at a time, the computation for each layer can be described as a matrix-vector multiplication. Consequently, the main bulk of the computation in an MLP requires successively performing matrix-vector multiplications with the layers' weights w^l and the previous layer's activations $\mathbf{a^{l-1}}$.

Convolutional Neural Networks (CNNs) provide a way of reducing the number of parameters in the network while also exploiting the spatial property in images where pixels located close together are more related. Figure 2.5 illustrates an example of the structure of a CNN.

Let x be a square input image with height and width W and a depth of D. For example, an input image might have 32×32 pixels where each pixel has a Red, Green, Blue (RGB) component hence having $D = 3$. The convolution operation partitions the image into small overlapping sections of $N \times N$ pixels where typically $N \ll W$. On each of these sections, F different filters are applied and the results stacked together giving an output image of depth F. This can be formalised as given an input image $x \in \mathbb{R}^{W \times W \times D}$, the convolutional weights $w \in \mathbb{R}^{N \times N \times D \times F}$ apply a transformation to give an output image $y \in \mathbb{R}^{W \times W \times F}$. The calculation of the convolution operation is given as

$$y_{i,j,f} = \sum_{q=0}^{N} \sum_{r=0}^{N} \sum_{s=0}^{D} x_{a,b,s} w_{q,r,s,f} \tag{2.17}$$

where $a = i + q - \lfloor N/2 \rfloor$ and $b = j + r - \lfloor N/2 \rfloor$ and assuming the image is padded with zeros around the borders. Expanding the sums in Equation 2.17 the input x can be transformed into a vector of length $N \times N \times D$ for a chosen i and j denoted $\mathbf{X}_{i,j}$. The weights are independent of i and j and are rearranged as a matrix of constants with $N \times N \times D$ columns and F rows to produce the output vector $\mathbf{y}_{i,j}$ of length F. This can be written as

$$\mathbf{y}_{i,j} = \mathbf{X}_{i,j} \mathbf{w} \tag{2.18}$$

which is a matrix-vector operation for each pixel of the image. An important property of the convolution that is exploited in this thesis is that the result for the convolution only depends on a small area around the coordinates i, j and that the weights, $\mathbf{w}$, are constant in regard to the chosen coordinates.

The art of how to arrange these layers and training them together is a rapidly developing area of research.

2.2.1 Low Precision Networks

CNNs come in a variety of flavours and can be highly customized to suit each application domain. While CNNs are typically trained with single/double floating-point precision, inference can be done with reduced precision and/or pruning without a significant loss

in accuracy, which opens up opportunities for building accelerators for deployment. Ternarization of weights [2, 48] is a popular quantization choice, as it delivers a significant reduction in the memory footprint of the CNN model, while also preserving the model accuracy to a large degree. In Ternary Weights Networks (TWNs) [48], parameters are quantized during training as follows:

$$W_i^l = \begin{cases} +s, & \text{if } W_i^l > \Delta \\ 0, & \text{if } |W_i^l| \leq \Delta \\ -s, & \text{if } W_i^l < -\Delta \end{cases} \tag{2.19}$$

where W_i^l are the parameters in each layer of the network, s is some scaling value and Δ is a positive threshold parameter used for quantization.

Interest in low precision CNNs has dramatically increased in recent years due to research which has shown that a similar accuracy to floating-point can be achieved [7, 15, 18, 57, 71, 92]. Due to the high computational requirements of CNNs, reduced precision implementations offer opportunities to reduce hardware costs and training times. Since FPGAs can implement arbitrary precision datapaths, they have some advantages over the byte addressable GPUs and CPUs for these applications. Moreover, the highest throughput implementations on all platforms utilise reduced precision for a more efficient implementation. In ternary neural networks (TNNs), implemented similarly to Li et al. [48], the value of the weights is restricted to be $-s$, 0 or s, where s is a scaling factor. This transforms Equation 2.16 to

$$a_i = f(s^l \mathbf{w_i^l} \mathbf{x^l} + b_i^l) \tag{2.20}$$

where $w_{ij}^l \in \{-1, 0, 1\}$ are the elements of $\mathbf{w_i}$, b_i^l is the bias term and $s^l \in \mathbb{R}$.

Applying it to the convolution operation in Equation 2.18, it can be rewritten with ternary weights $\mathbf{t}$ and scaling factor s as

$$\mathbf{y}_{i,j} = s\mathbf{X}_{i,j}\mathbf{t} \tag{2.21}$$

The matrix of values for the ternary weights, $\mathbf{t}$, are constant, restricted to $-1, 0, 1$ and known after training. The input $\mathbf{X}_{i,j}$ is an unknown dense vector and s is a constant scalar as defined above. Therefore a hardware block can be designed to take $\mathbf{X}_{i,j}$ as input and output $\mathbf{y}_{i,j}$. With these assumptions, Chapter 4 will demonstrate that a hardware block can be implemented with parallel adder trees to skip a large portion of the computation. The consideration of the ternary convolutional operation in this manner is a core contribution of this thesis and the benefits of this approach particularly in regard to unstructured sparsity are demonstrated in Chapter 4. These methods are applied in two case studies in Chapter 5.

2.3 Problem Domains

2.3.1 Datasets for Benchmarking KAFs

The datasets chosen to benchmark for classification and novelty detection were an artificial dataset generated using *sklearn.make_classification* [65] and the mlbench dataset *Satellite* [47]. The regression datasets were an artificial dataset generated using *sklearn.make_regression* and the UCI *Combined Cycle Power Plant* dataset from [82].

These datasets were chosen as they are readily available for anyone to reproduce the results. As this thesis is more focused on the hardware implementation rather than measuring the performance on NORMA itself, these are sufficient to monitor quantization effects.

2.3.2 CIFAR10

The CIFAR10 dataset contains 50000 training and 10000 test images from 10 different categories. These categories are: airplane, automobile, bird, cat, deer, dog, frog, horse, ship and truck. The square images are in RGB colour and are reasonably small, 32×32 pixels. This dataset is widely used to benchmark image classification techniques.

2.3.3 Automatic Modulation Classification

Automatic Modulation Classification (AMC) is a common requirement in Cognitive Radio Networks (CRN) for both military and civilian applications. For example, AMC plays a pivotal role in dynamic spectrum management in CRNs, where cognitive radios are able to detect idle frequency bands and transmit data in these bands such that the primary users of these channels are not affected [70]. AMC, however, can be a challenging task for two reasons: (1) often, there is no a priori information known about the transmission signal/channel (*e.g.* signal-to-noise ratio, carrier frequency, etc), especially in military contexts, and (2) latency/throughput requirements to achieve real-time processing often limit the complexity of the implementation and its runtime effectiveness.

The classical approach to AMC is built on statistical/stochastic methods that require high domain expertise and frequent manual tuning to achieve reliable performance in each specific environment. Popular methods include maximum likelihood hypothesis testing [63], and feature-based classification using high-order cumulants [1], or cyclo-stationary features [70]. Notable recent efforts have been made into using deep learning techniques to push this field further [62, 73, 93], but while these studies have demonstrated an improvement to classification accuracy and reliability, the practicality of realizing a real-time machine-learning based AMC implementation is left unexplored. Section 5.2 adopts a more holistic approach, where expertise in both machine-learning and hardware design is used in tandem to set a benchmark for real-time automatic modulation classification.

2.4 Summary

This chapter introduced the theory for several machine learning techniques. Some key concepts used for kernel methods were defined such as the kernel function, the dictionary and the weights. It examined the formulation of NORMA, an online KAF, in detail. Chapter 3 will propose a method to implement NORMA on an FPGA.

Neural Networks were also described in this chapter. Through a redefinition of the convolution, Equation 2.17 was transformed into a matrix vector operation in Equation 2.18. Section 2.2.1 explored low precision networks, in particular the use of ternary weights. Chapter 4 and 5 will consider TNNs in more detail and propose techniques to implement them on FPGAs.

Chapter 3

Online Kernel Methods in Hardware

This chapter will examine computation dependencies and propose a low latency hardware implementation of an online kernel method. The work in this chapter was published in FPT-15 as *Braiding: A scheme for resolving hazards in kernel adaptive filters* [80].

As introduced in Chapter 2, kernel methods create a decision boundary using the function:

$$f(x) = \sum_{i=1}^{D} \alpha_i \kappa(x, d_i) \quad , \tag{3.1}$$

This is the main component of the computation for inference in kernel methods. In algorithms such as SVM the training and inference portions are distinctly separate. The SVM may train for several hours and, after this is completed, this trained model will be used for inference without considering any new information. In online learning methods such as KAFs, the inference and training computations are typically combined. As a new example arrives, a KAF makes a prediction or inference based on previous examples. This prediction can be used in the application. The KAF then considers this new example and determines how to update itself so that it will improve its predictions for future examples. When considering hardware acceleration, the application determines many factors in the implementation.

There are a few different goals when considering hardware acceleration of machine learning. In some cases, the purpose is designing a general accelerator for training. These applications address the long training times that is typical in machine learning, especially on large datasets. In others, the training time is relatively insignificant when a simple model has to be deployed on a wide scale. In this case, the inference

time matters a lot more to provide fast response times and provide high throughput [36]. This chapter considers online kernel methods so latency and throughput for both training and inference is critical.

Due to the nature of online kernel methods, the previous training iteration must be completed before the inference can begin. Dependencies such as this can cause significant difficulties in high performance applications. A task is said to be dependent on another task if it must wait for it to be completed before another one can begin. If dependencies are not managed properly, they can cripple the performance of highly parallel systems. Of particular note is the presence of dependencies in algorithms. Dependencies can require portions of the system to stall while waiting for a previous task to finish to use that result. Careful scheduling of tasks attempts to minimize the amount of time stalled to efficiently use all the computation available.

In single threaded systems, managing dependencies is not as much of an issue as there is always useful work to be done. However, using only a single thread for an application can seriously limit the speed at which it can be computed. Moving to a system with multiple computational cores available offers the potential to significantly improve the performance. Unfortunately this introduces a range of issues into the system in an effort to keep all the available resources doing useful work. Resolving these algorithmic dependencies to achieve high performance has a long history in computation. It is still a very active research area in CPU design and good solutions can improve performance significantly.

The NORMA algorithm, discussed in Chapter 2 has a strong dependency inside its update step. As an online learning algorithm it considers each example as it arrives before choosing to update the model. The model has to finish updating before the next example can be computed.

For an FPGA implementation, this dependency is an issue for high performance applications. In addition, the original algorithm has components that normally require

floating point operations, such as the exponential. This chapter demonstrates a technique to mitigate these issues.

When implementing an algorithm in hardware, the amount of resources used for each component of an algorithm can easily be varied. Assuming the goal is only minimizing latency, the entire algorithm could be implemented as complex combinational logic with memory for state transitions. In the case of NORMA, the entire system of equations could be computed in a single, very slow clock cycle. The time taken for each example to pass through this long combinational logic path would be small but result in a very low throughput. This is because each example has to wait for the entire hardware path to be available before beginning the next one. Including pipelining stages into this path may increase the latency slightly but can provide significant improvements in throughput. The goal of this chapter is to achieve a low latency implementation of NORMA with a reasonable throughput. This is achieved with careful pipelining of the algorithm accounting for algorithmic dependencies.

FPGA-based implementations of KAFs have previously been proposed to achieve higher throughput, lower latency or decreased power usage. Matsubara et al. [56] described an implementation of the least mean square (LMS) algorithm which computed a solution ignoring data hazards, and then applied correction terms. A soft vector processor optimised for implementation of the KRLS algorithm was reported by Pang et al. [64], this having the advantage that different KAFs can be implemented with only software changes. Ren et al. [72] implemented a simpler algorithm, quantized kernel least mean squares (QKLMS) [12], which has many similarities to NORMA. Their approach computed the update sequentially, leading to small area, however the design described in this work has 50× higher throughput.

The highest throughput KAF architecture reported to date is an implementation of kernel normalized least mean squares (KNLMS)[23]. The design presented in this chapter achieves a lower throughput however does not require multiple parallel problems. This allows application beyond parameter search or parallel problem domains.

3.1 Hardware Architecture of NORMA

This section aims to create a high throughput and low latency implementation of NORMA. The formulation of NORMA was discussed in Chapter 2 was given as:

$$(\alpha_i, \alpha_t, b, \rho) = \begin{cases} (\Omega\alpha_i, 0, b, \rho + \eta\nu) \text{ if } y(f(x) + b) \geq \rho \\ (\Omega\alpha_i, \eta y, b + \eta y, \rho - \eta(1 - \nu)) \text{ otherwise} \end{cases} \tag{3.2}$$

$$(\alpha_i, \alpha_t, \rho) = \begin{cases} (\Omega\alpha_i, 0, \rho + \eta\nu) \text{ if } f(x) \geq \rho \\ (\Omega\alpha_i, \eta, \rho - \eta(1 - \nu)) \text{ otherwise} \end{cases} \tag{3.3}$$

$$(\alpha_i, \alpha_t, \epsilon) = \begin{cases} (\Omega\alpha_i, 0, \epsilon - \eta\nu) \text{ if } |y - f(x)| \leq \epsilon \\ (\Omega\alpha_i, \delta\eta, \epsilon + \eta(1 - \nu)) \text{ otherwise} \end{cases} \tag{3.4}$$

where $\delta = sign(y - f(x))$, $\Omega \leq 1$ and $f(x) = \sum \alpha_i \kappa(x, d_i)$.

Examining equations 3.2, 3.3 and 3.4 three properties of NORMAs formulation can be extracted:

(1) NORMA is a sliding window algorithm
(2) each iteration, the weights decrease by a multiplicative factor ($\alpha_i \rightarrow \Omega\alpha_i$); and
(3) the computational cost of the update is small compared to the evaluation of the decision function $f_t(x_{t+1})$.

The first property is determined by considering examples gradually added to the dictionary with the same initial weight magnitude. This slowly builds up the dictionary one new example at a time. To stop this growing indefinitely, some maximum number of dictionary entries are needed meaning, once its full, some examples need to be removed. Choosing the oldest as the least relevant examples leads to a sliding window. This first property allows the decision function to be expressed as a combination of the examples currently in the pipeline, x_i, and a previous dictionary, $\hat{\mathcal{D}}$.

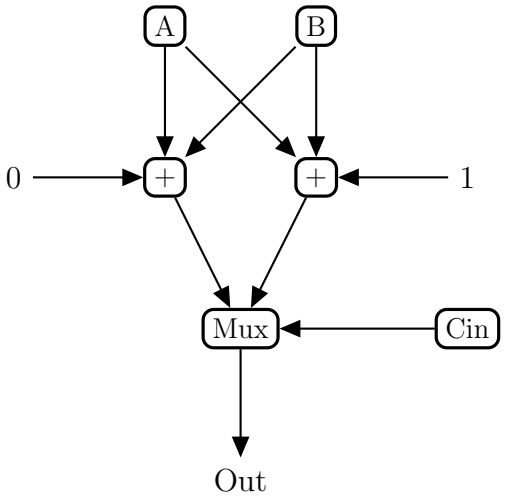


Figure 3.1: Carry Select Adder Example

As shown in equations 3.2, 3.3 and 3.4, they all multiply the weights, α_i, by a factor of $\Omega \leq 1$. This second property provides a simple relationship to update the weights for the final dictionary reducing the computation to be completed each clock cycle.

The final property is necessary as it determines the critical path and hence the effectiveness of pipelining as one update is computed each clock cycle for a fully pipelined design. This is evident from the update expressions above. The decision function, $f(x)$, has to compute the kernel function with every example in the dictionary. The update expressions however are just simple scalar operations.

By using *braiding*, a technique proposed in this section, $f_t(x_{t+1}) = \sum_{i=1}^{D} \alpha_i \kappa(x_{t+1}, d_i)$ can be rewritten to include a single pipeline stage as a function of the dictionary in the previous clock cycle ($\hat{\mathcal{D}}$). This concept is similar to a the hardware structure of a carry select adder. Figure 3.1 demonstrates the idea behind a carry select adder. In this case, adders are used to compute the results of $A + B$ with a carry in of 0 or 1. When the carry is known, the correct result can be selected and sent as output.

In the case of braiding, $f_t(x_{t+1})$ can be separated in a similar way. Consider the case where it is unknown if the previous example was added to the dictionary or not. If it was added, then the dictionary should shift out the oldest example. Otherwise, the

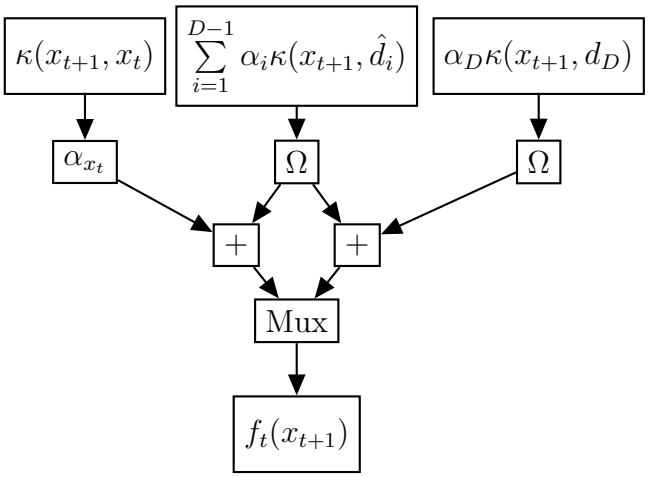


Figure 3.2: Hardware representation of equation 3.5

oldest example should still be a component of f_t. More formally, this can be expressed as the equation:

$$f_t(x_{t+1}) = \sum_{i=1}^{D-1} \Omega \alpha_i \kappa(x_{t+1}, \hat{d}_i) \tag{3.5}$$
$$q \begin{cases} \\ \end{cases} + \begin{cases} 0 \text{ if } x_t \text{ is not added} \\ \alpha_{x_t} \kappa(x_{t+1}, x_t) \text{ otherwise} \end{cases}$$
$$+ \sum_{i=D}^{D-q} \Omega \alpha_i \kappa(x_{t+1}, \hat{d}_i)$$

where q is 1 if x_t is added to the dictionary and 0 otherwise, Ω is the forgetting factor, D is the size of the dictionary and x_t is the example ahead in the pipeline. When the example x_t is added, the sliding window shifts to include this new example and discards the oldest example in the dictionary. In this case $q = 1$ meaning the third term in equation 3.5 is 0. If x_t is not added, the dictionary remains unchanged and the third term includes the oldest dictionary entry.

Figure 3.2 shows a hardware datapath which implements equation 3.5. In a similar way to the carry select adder, two results are computed but only one is used. In this case, a single pipelining stage has been added in the computation compared with the original

equation, $f_t(x_{t+1}) = \sum_{i=1}^{D} \alpha_i \kappa(x_{t+1}, d_i)$. Computing the terms at the top of Figure 3.2 can be done on a previous clock cycle which improves the throughput marginally. In this figure, the multiplexer chooses whether to add x_t to the dictionary, or discard it so that the oldest dictionary entry can now be merged into the sum. This design is analogous to braiding as the core of the sliding window has either $\kappa(x_{t+1}, x_t)$ or $\alpha_i \kappa(x_{t+1}, \hat{d}_i)$ braided into it each clock cycle. The amount of computation required for the inputs of Figure 3.2 is significantly higher than the computation shown in the figure. Fortunately, the braiding technique can be expanded to an arbitrarily pipelined implementation as expressed in equation 3.6.

$$
\begin{aligned}
f_t(x_{t+1}) = & \sum_{i=1}^{D-p} \Omega^p \alpha_i \kappa(x_{t+1}, \hat{d}_i) \\
& q \left\{ \begin{array}{l}
+ \begin{cases} 0 \text{ if } x_{t+1-p} \text{ is not added} \\ \Omega^{p-1} \alpha_{x_{t+1-p}} \kappa(x_{t+1}, x_{t+1-p}) \text{ otherwise} \end{cases} \\
+ \begin{cases} 0 \text{ if } x_{t+2-p} \text{ is not added} \\ \Omega^{p-2} \alpha_{x_{t+2-p}} \kappa(x_{t+1}, x_{t+2-p}) \text{ otherwise} \end{cases} \\
\qquad \vdots \\
+ \begin{cases} 0 \text{ if } x_t \text{ is not added} \\ \alpha_{x_t} \kappa(x_{t+1}, x_t) \text{ otherwise} \end{cases}
\end{array} \right. \\
& + \sum_{i=D-p+1}^{D-q} \Omega^p \alpha_i \kappa(x_{t+1}, \hat{d}_i)
\end{aligned}
\tag{3.6}
$$

In this case, q is the number of examples in the pipeline added to the dictionary, p is the number of pipeline stages and $\hat{d}_i$ are examples in the dictionary at time $t - p$. The first term is the portion of the decision function that depends on recent examples in the sliding window such that in p cycles time they are guaranteed to still be in the dictionary and hence must be used in the computation of $f_t(x_{t+1})$. The dependence of the decision function on the examples currently in the pipeline is expressed in the second

component. Using the shift of q, the final term in equation 3.6 sums the contributions from the uncertain and oldest portion of the dictionary.

For a practical hardware implementation, the computation of $f_t(x_{t+1})$ is separated into two phases. The first is the computation of the kernel function, $\kappa(x_{t+1}, \hat{d}_i)$ needed for all parts of Equation 3.6. This involves $D + p$ parallel kernel evaluation blocks to compute this function for each of the examples in the dictionary and pipeline. Following this is a multiplication and sum of $\alpha_i \kappa_i$ again needed for all parts of the equation. This completes the computation of $f_t(x_{t+1})$ which is then passed to the update section to determine if the example is to be added.

3.1.1 Pipelining

Figure 3.3 shows a high level view of the proposed pipeline for the online training of the NORMA algorithm implementing equation 3.6. This pipeline assumes that $D > p = k + s + 1$ using the notation for p, k, s defined in Figure 3.3. This is reasonable because the number of cycles for a single kernel evaluation, k, depends only on the number of features and, using a method such as an adder tree, s, scales as $O(\log_2(D))$. The three stages of the pipeline vary greatly in terms of the hardware requirements for their implementation. The kernel evaluation scales as $O(F(D + p))$ where F is the number of features or dimensionality of the input, followed by the summation stage which is $O(D)$ additions and multiplications. The final stage that computes the update is $O(1)$ in hardware and time. However, as the update must take place in a single cycle to avoid data hazards, this final stage is the critical path restricting the clock frequency.

The kernel evaluation must begin by calculating $\kappa(x_{t+1}, \cdot)$ with the dictionary entries and all the examples in the pipeline as any of them could be needed in the computation of $f_t(x_{t+1})$. As x_{t+1} progresses along the pipeline, the possible combinations of dictionary entries are constrained. At each clock cycle, either the example at the final stage of the pipeline is discarded or the oldest example in the dictionary. Hence, each stage of the pipeline in the kernel evaluation contains one less example than the previous. After k

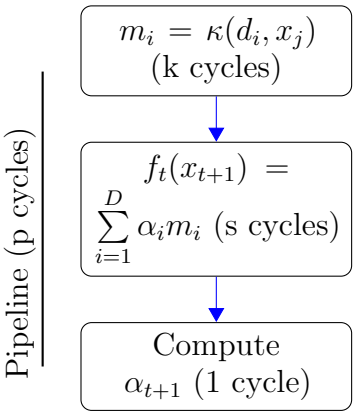


Figure 3.3: The pipelined calculation

cycles, this will result in $D + p - k = D + s + 1$ values to propagate to the summation stage. These scalar values that are the result of the kernel evaluation are given the label of z_i for the pipelined examples and v_i for the dictionary examples. With reference to Equation 3.6, v_i represents the terms in the top and bottom summations where as z_i are the results needed for the examples that might yet be added.

The kernel evaluation stage outputs $D + s + 1$ scalar values which are the results of evaluating the kernel function of an example with the current dictionary and the $s + 1$ examples ahead in the pipeline. As shown in Figure 3.3, this is followed by the summation stage of $\sum \alpha_i z_i + \sum \alpha_i v_i$ where z_i is defined as the result of the kernel evaluation for pipelined examples and v_i for the dictionary entries. In terms of Equation 3.6, z_i are the evaluations of the kernel function for terms within the q bracket and v_i are all other kernel evaluations. The summation stage does not use all values of z_i and v_i, discarding one each cycle as in equation 3.6. Additionally, for the examples ahead in the pipeline, the values of α_i have yet to be calculated. This requires the summation to be split into sum left ($Sum_L = \sum \alpha_i z_i$) and sum right ($Sum_R = \sum \alpha_i v_i$).

Sum_L is the summation of terms ahead in the pipeline shown in Figure 3.4 or the q terms in equation 3.6. This diagram shows the cumulative sum spread over $s - 1$ cycles to first multiply the kernel evaluation with freshly computed α values and then added it to a running total of Sum_L which is initialized to 0 for the first cycle. Ω is the forgetting factor which is multiplied into the sum each clock cycle. The 'Mux' in

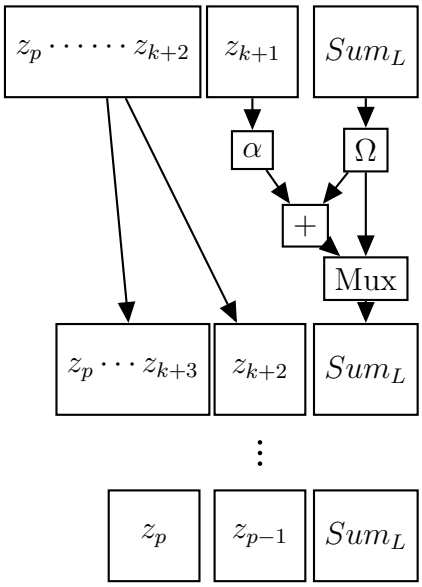


FIGURE 3.4: The pipeline for Sum_L

Figure 3.4 and in all other figures in this chapter refer to a multiplexor with two inputs and the control line connected to the decision of whether or not to add an example to the dictionary. Sum_L finishes with two remaining z values and the sum giving a total of three output values into the final stage of the sum. The forgetting factor is multiplied into Sum_L each clock cycle.

Sum_R differs from Sum_L as all the α values are readily available from the current dictionary. In the first stage the value $w_i = \alpha_i v_i$ is calculated for all examples followed by a pipelined adder tree to sum w_i. However, all w_i values cannot be summed immediately as some may be evicted from the dictionary and hence give the incorrect result for Sum_R. Under the assumption that the oldest example is removed, the examples in the dictionary that have an index less than $D-s-1$ or the examples that are 'young enough' are guaranteed to be in the final sum and therefore can be added in parallel. The other w_i values above this threshold index are questionable as they are in range of being shifted out of the dictionary by the sliding window. As they cannot be immediately added into Sum_R they are *braided* into the sum each clock cycle when it is determined that it is impossible for that dictionary entry to be shifted out before the calculation $f_t(x_{t+1})$ completes.

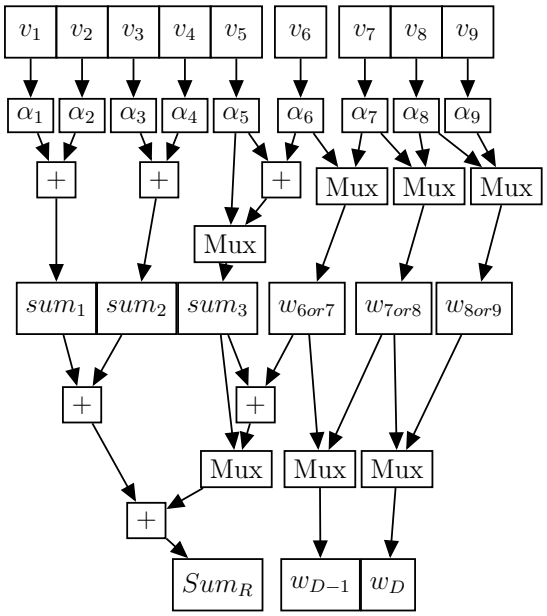


Figure 3.5: An example 3 stage pipelined braiding sum

The term braiding is used to describe this structure as each clock cycle another term is combined into the sum analogous to a hair or rope braid. Figure 3.5 shows an example of Sum_R for a dictionary size of 9 with 3 pipeline stages. Section 3 of the appendix goes through a numerical example for this figure. v_1 to v_5 are the 'young enough' examples as within the next 4 clock cycles they are a component in the computation of $f_t(x_{t+1})$. These examples are also referred to as the core of the sliding window in this chapter. v_6 to v_9 are uncertain as in 4 clock cycles they could all be removed if all the pipelined examples are added. In the first stage in the figure if an example at the end of the pipeline is added v_9 is discarded from the dictionary leaving three uncertain examples. Conversely, if it is not added then example v_6 is *braided* into the sum also leaving three uncertain examples. These are passed along to the next stage while the others are summed in an adder tree with braiding until only two unbraided examples remain.

The final stage of the summation combines Sum_L $(= \sum \alpha_i z_i)$ and Sum_R $(= \sum \alpha_i v_i)$ and the computation of α is shown in Figure 3.6. In Figure 3.5 the forgetting factor was temporarily ignored. Instead it is computed separately over $s-1$ clock cycles so when Sum_L and Sum_R are combined, this value of Ω^{s-1} is used to compensate as $\sum \Omega^{s-1}\alpha_i \cdot v_i = \Omega^{s-1} \sum \alpha_i \cdot v_i$. Furthermore, to account for an additional cycle, Ω is

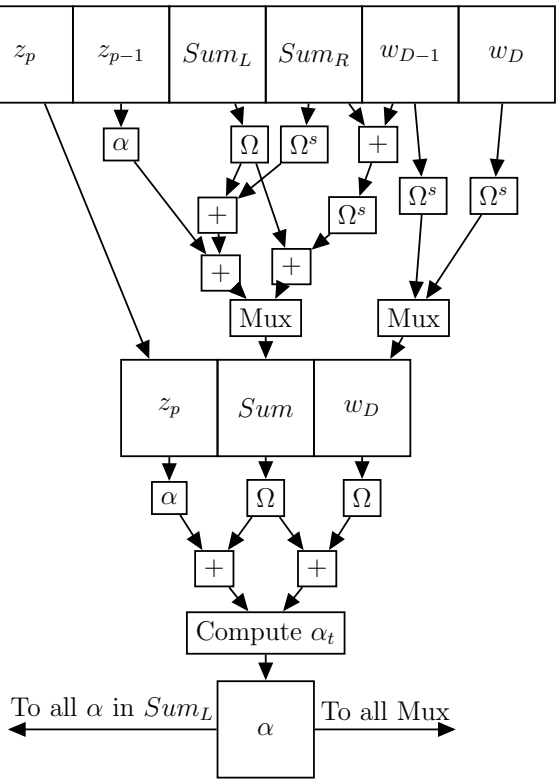


FIGURE 3.6: Data path of the last stage of sum and the update computation

applied once more so the factor becomes Ω^s. Sum_R and Sum_L are then combined and the remaining task is to incorporate a single dictionary entry (w_D) and pipeline example (z_p) in the sum. In the final cycle, the value of $f_t(x_{t+1})$ can be computed. This is passed into the block $\alpha(f_t(x_{t+1}))$ which computes the weight α based on the application. α is then forwarded to the stages in Figure 3.6 and to the calculation of Sum_L. During the calculation of α, whether to add an example to the dictionary or not must be determined. The result of this decision is passed to the select lines of all the multiplexers on the next clock cycle.

3.1.2 Fixed Point

To reduce hardware usage and latency on the critical path involving computing the update, the implementation uses fixed point arithmetic. Fixed point allows the representation of decimal numbers by a 'hidden' multiplication with $2^{-(\#of fractionalbits)}$. In

hardware, all the operations are done as though the bits represent an integer other than the occasional bit shifting. However, they are interpreted as decimal numbers generating higher precision with minimal overhead. Fixed point is used in this design to dramatically reduce the hardware cost compared to floating point arithmetic. Compared to floating point, fixed point requires careful consideration of the precision and range to avoid issues such as overflow and underflow. Consideration of NORMA is needed to determine the precision and range required.

As the weights and output of the kernel evaluation are tightly bounded to be scalar values less than a magnitude of 1, the impact of fixed point on the accuracy of the algorithm should not be severe if sufficient precision is used. The update step of NORMA is examined to assess the impact of a fixed point implementation with 12 fractional bits. It is characterized by an initial value η multiplied by the forgetting factor Ω each clock cycle. Under the assumption that all examples are added leads to the maximum value of the oldest dictionary weight being $\eta\Omega^{D-1}$. Replacing Ω with $1-\eta$ for classification and novelty detection [39] leads to the calculation of $\eta = \frac{1}{D}$ to obtain the maximum value for the last dictionary entry. This results in $\frac{1}{D}(1-\frac{1}{D})^{D-1}$ which for 12 fractional bits and $D = 200$ is $0.00184 = \frac{7.55}{2^{12}}$. The more realistic assumption that half the examples are added further reduces this to $\frac{2.77}{2^{12}}$. Using fixed point multiplication with truncation erodes this further demonstrating that the use of fixed point can restrict the effective dictionary size of NORMA when insufficient precision is used. A brute force search for the maximum dictionary for each possible fractional width (f_w) was performed under the assumption that every example is added. From these results the expression $D \leq 3 * 2^{\frac{f_w-3}{2}}$ approximates the maximum size of the dictionary when using fixed point. There is little point in having a larger dictionary than this as all older dictionary entries will have a weight of zero. In reality, the size of the dictionary should be much smaller than this bound as η likely will not be optimal for the dictionary size and the assumption that all examples are added is strong. To achieve reasonable utilization from the dictionary it should be quite a bit smaller than this bound.

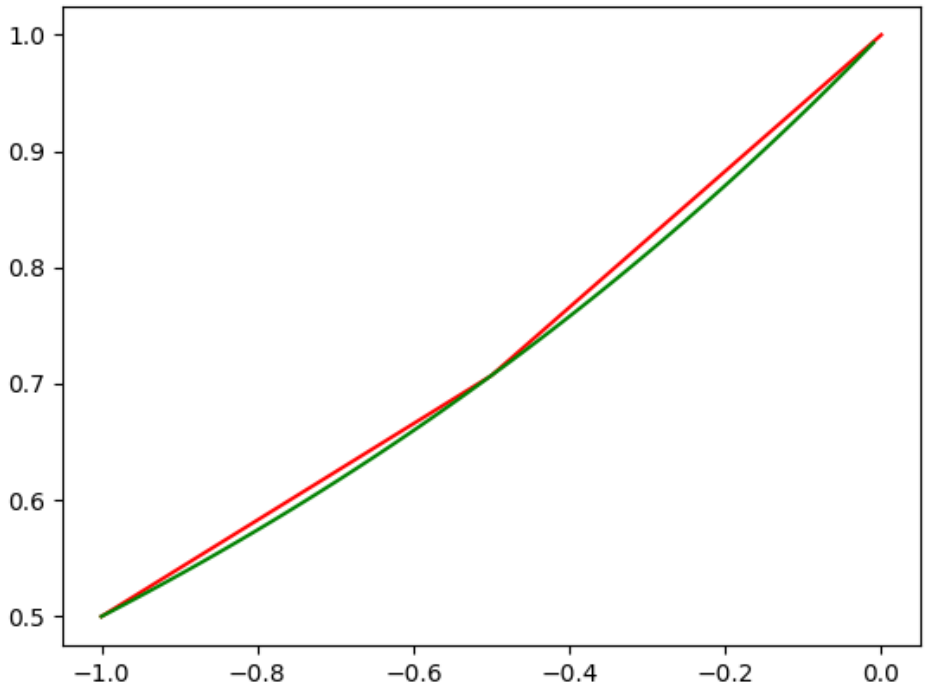


Figure 3.7: Linear interpolation with two lines for 2^x

Without loss of generality, the Gaussian kernel, $\kappa(x_i, x_j) = e^{-\gamma\|x_i - x_j\|_2^2}$, was implemented in this design since it is a commonly used kernel. The fixed point implementation uses the property that $e^{bx} = 2^{ax}$ for a given b [88]. The value of ax is split into an integer and fractional component, giving $2^{ax} = 2^{int(ax)}2^{frac(ax)}$ where $-1 \leq frac(ax) \leq 0$. Using the restriction that $ax \leq 0$ for the Gaussian kernel allows the value of $2^{frac(ax)}$ to be calculated using a lookup table with linear interpolation between -1 and 0. This is demonstrated in Figure 3.7 with two lines approximating the curve. The values for these lines, m and b, in the equation $y = mx + b$ are stored in the lookup table. Based on the value of x, the correct line is selected and used to approximate the value of $2^{frac(ax)}$.

2^{ax} is then obtained by bit shifting this result with the integer component, $int(ax)$. This allows the exponentiation to be computed in 5 operations or a couple of clock cycles while the calculation of $\|x - y\|_2^2$ depends on the size of the adder tree constrained

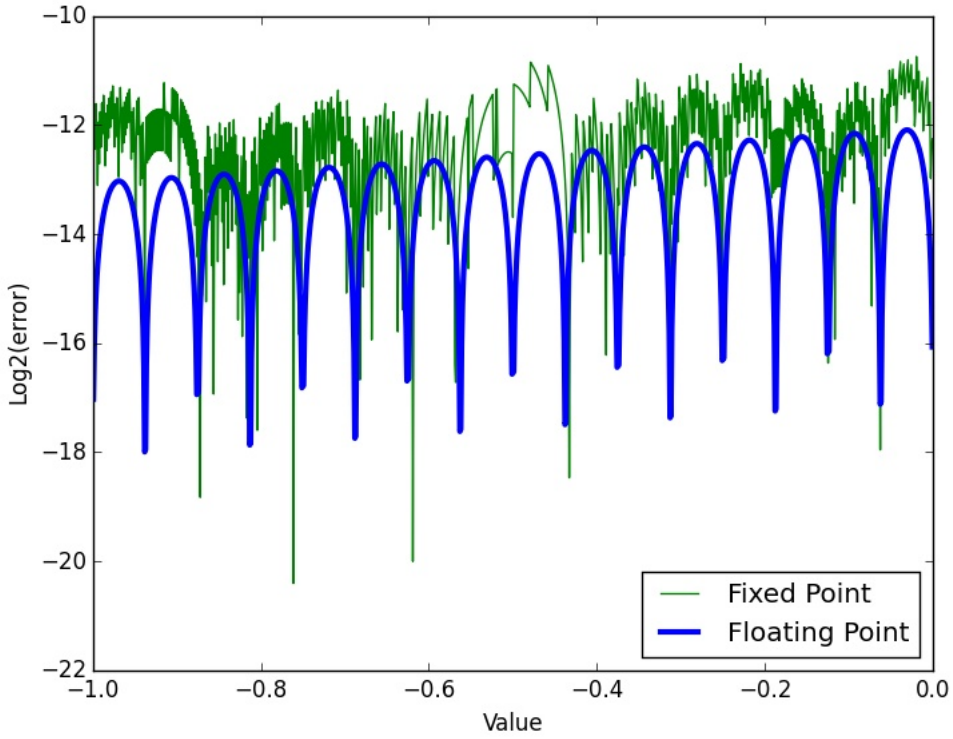


Figure 3.8: Approximation error of a 16 value lookup table

by the number of features used. Figure 3.8 shows the error associated with using this method. The two traces represent linear interpolation computed in floating-point and fixed point with fractional width of 12. For a 16 value lookup table and 12 fractional bits, the error only affects the least significant one or two bits.

3.2 Results and Discussion

The proposed architecture was implemented using CHISEL [4] with an open source implementation available at github.com/da-steve101/chisel-pipelined-olk.git. A modified Kernlab implementation of NORMA for use with the Classification And REgression Training (CARET) package is used to obtain the baseline floating-point learning performance of NORMA on various datasets. A C implementation of NORMA was created to measure the speed of a CPU implementation for comparison. This

is also available in the repository. All other datasets and scripts used to obtain the results in this chapter are available in the repository with instructions in the README. The code was synthesised for a Xilinx Virtex-7 XC7VX485T-2FFG1761C on a VC707 development board with speed grade 2. No interface was implemented, and the following analysis assumes that a streaming interface can provide enough throughput to keep the system full.

The resource usage and clock frequency was obtained for a varying dictionary size with $F = 8$ features for the $NORMA_n$ application are summarised in Table 3.1. This table shows the growth in resource usage with dictionary size and fixed point precision. From this table it is clear that the number of DSP's is the bottleneck for this architecture. This is because approximately $D * (F + 1)$ multiplications are required for the kernel evaluation. The number of DSP's used also grows with the bitwidth as the full result can no longer be computed in a single DSP for more than 18 bits. An increasing fixed point bitwidth also results in a longer delay in the multiplication stage, reducing the maximum clock frequency also shown in the table.

Tables 3.2, 3.3 and 3.4 compare fixed point implementations with the Kernlab R implementation [37] as a floating-point reference. The caret package [43] facilitated this comparison, and parameters for the floating-point implementation were chosen using a parameter search with 10 fold cross validation [41]. Training was conducted using the H and Area Under the Receiver Operating Characteristic Curve (AUC) measures for classification and novelty detection, and the L1 and L2 error for regression. The AUC and H measures were chosen as they provide a better overview of the classifier or novelty detector than accuracy [30]. Both AUC and H measures are used as there is a lack of consensus on which should be used [20]. An optimal classifier has $AUC = 1$ and $H = 1$ whereas a classifier that randomly guesses should have an $AUC = 0.5$ and $H = 0$. The fixed point implementation was then tested using the same parameters found with the floating-point cross validation to assess its impact. A lookup table with 16 points as shown in Figure 3.8 was also used unless otherwise stated. In Tables 3.2,

Table 3.1: Performance of $NORMA_n$ for F = 8 and Lookup Table = 16 points on a Virtex 7

D =	16	32	64	128	200
Fixed 8.10					
DSPs (/2800)	309	514	911	1679	2556
Freq (MHz)	133.0	137.8	137.4	131.0	127.3
Latency (clocks)	10	11	12	12	13
Slices (/75900)	4615	8194	14663	29113	46443
Fixed 8.16					
DSPs (/2800)	595	988	1749	2800	*
Freq (MHz)	115.2	114.1	93.2	98.2	*
Latency (clocks)	10	11	12	12	*
Slices (/75900)	6188	11622	20512	58889	*
Fixed 8.22					
DSPs (/2800)	1236	2056	*	*	*
Freq (MHz)	101.2	97.29	*	*	*
Latency (clocks)	10	11	*	*	*
Slices (/75900)	10971	18976	*	*	*
Fixed 8.28					
DSPs (/2800)	1236	2056	*	*	*
Freq (MHz)	97.2	89.8	*	*	*
Latency (clocks)	10	11	*	*	*
Slices (/75900)	13819	23931	*	*	*

3.3 and 3.4, i_w denotes the integer width that was used. This was determined by finding the minimum value before overflow occurs.

The datasets chosen to benchmark for classification and novelty detection were an artificial dataset generated using *sklearn.make_classification* [65] and the mlbench dataset *Satellite* [47]. The parameters chosen for the artificial dataset were *n_samples* = 1000, *n_features* = 8, *n_informative* = 4, *n_redundant* = 2 and *random_state* = 101. For the *Satellite* dataset, the multiclass problem was simplified to a two class problem by taking the largest class out and combining the others. The novelty detector for this dataset trains on one of the classes in the artificial dataset and for the satellite dataset it trains on the combined classes to detect the 'anomalous' red soil class. The regression datasets were an artificial dataset generated using *sklearn.make_regression* and the UCI *Combined Cycle Power Plant* dataset from [82]. The parameters chosen for the artificial dataset were *n_samples* = 1000, *n_features* = 8, *n_informative* = 6 and

Table 3.2: Results for $NORMA_c$ in Fixed vs Floating point

Dataset	Artificial ($i_w = 7$)		Satellite ($i_w = 8$)	
Measure	AUC	H	AUC	H
Floating $NORMA_c$	0.893	0.550	0.995	0.947
Fixed 18 bits	0.618	0.108	0.673	0.145
Fixed 24 bits	0.745	0.255	0.996	0.922
Fixed 30 bits	0.899	0.579	0.998	0.954
Fixed 36 bits	0.903	0.586	0.997	0.959

Table 3.3: Results for $NORMA_n$ in Fixed vs Floating point

Dataset	Artificial ($i_w = 8$)		Satellite ($i_w = 8$)	
Measure	AUC	H	AUC	H
Floating $NORMA_n$	0.641	0.140	0.836	0.358
Fixed 18 bits	0.664	0.174	0.5	0
Fixed 24 bits	0.658	0.162	0.503	0.130
Fixed 30 bits	0.658	0.162	0.800	0.295
Fixed 36 bits	0.658	0.162	0.825	0.348

Table 3.4: Results for $NORMA_r$ in Fixed vs Floating point

Dataset	Artificial ($i_w = 6$)		Combined Cycle Power Plant Dataset ($i_w = 6$)	
Measure	L1	L2	L1	L2
Floating $NORMA_r$	0.773	0.934	0.700	0.697
Fixed 18 bits	0.760	0.899	0.637	0.621
Fixed 24 bits	0.776	0.939	0.666	0.643
Fixed 30 bits	0.777	0.943	0.653	0.628
Fixed 36 bits	0.777	0.943	0.653	0.628

Table 3.5: Single core CPU learning performance with F=8 compared with $NORMA_n$ using 8.10 fixed point

D =	16	32	64	128	200
Freq (MHz)	2.83	1.51	0.77	0.38	0.25
Latency/example (ns)	353.2	660.9	1293.2	2625.2	4025.8
FPGA Speedup (×)	47.0	91.3	178.44	344.7	509.2
Latency Reduction (×)	4.69	8.296	14.87	28.7	39.2

random_state = 101. All the datasets were then partitioned into a test and training set using the *createDataPartition* function in caret with an 80% split. The preprocessing functionality in caret was then used to center and scale the datasets using the training set with the exception of the artificial classification dataset. These four datasets are available from the github repository, together with four parameters files which enable the results to be reproduced.

Table 3.6: Impact of Gaussian Kernel Approximation (7.29 on artificial two class)

Table Size	1	2	4	8	16
AUC	0.907	0.906	0.904	0.903	0.903
H	0.588	0.590	0.586	0.586	0.586

The results in Tables 3.3 and 3.4 show that the use of fixed point does not have an impact on the learning performance of the algorithm for bitwidths greater than 18 bits. The cases in which fixed point actually improves the learning performance of the algorithm are attributed to noise in the results. For classification Table 3.2 shows that using fixed point with insufficient precision does have a detrimental impact on the results which is the expected case for general datasets. For implementations with similar fractional width precision to IEEE 754 (24 bits) there is no noticeable difference between fixed and floating-point implementations.

As the fixed point increases in precision, the learning performance is expected to improve. This is a trade off with the amount of hardware used hence affecting the maximum dictionary size as shown in Table 3.1. Due to fixed point changing the nature of the algorithm, performing the cross validation in fixed point may achieve better results than a cross validation in floating-point.

Table 3.1 shows the resource usage and clock frequency for NORMA with novelty detection in fixed point. Novelty detection has a shorter critical path in the update step compared with classification and regression resulting in a reduction in clock frequency of around 10-15 MHz compared to the other applications. The resource usage is however very similar as the majority of the calculation is the same.

Table 3.5 shows the results for a single core C implementation of $NORMA_n$ compiled using GCC 4.9.2 with '-O3' and run on an Intel i7-4510U @ 2.00GHz compared with the results in Table 3.1 for the 18 bit fixed point implementation. As this algorithm is very easily parallelized on a CPU, it can be reasonably assumed that N cores would achieve a speed up of N times. A GPU implementation is beyond the scope of this work. It is noted that, by increasing the number of features used (F), the speedup of the FPGA implementation improves dramatically as all additional features are calculated

in parallel with minimal impact on the clock frequency. However, this increases the amount of hardware needed for the implementation hence restricting the dictionary size.

Table 3.6 shows the effectiveness of the Gaussian kernel approximation. There is no noticeable difference in learning performance on the artificial two class dataset even when there is a single line from 0 to -1 that is shifted for integer powers. It is suggested that, due to the fixed initial value of the weights and the weight decay by Ω, NORMA is insensitive to the accuracy of the kernel function. The impact on other kernel methods could be investigated in future work.

The NORMA implementation in this chapter can be compared to the implementation of a KRLS vector processor designed for low latency machine learning [64]. While KRLS based algorithms have been shown to have superior learning performance to NORMA [84] the latency in the online learning is about two orders of magnitude lower with three orders of magnitude increase in throughput for the implementation described in this chapter. Compared with the approach in [72], which reports a sample rate of 2.4 MHz, braiding provides a $50\times$ increase in throughput. The algorithms are similar enough that a direct comparison is meaningful as the braiding technique could be applied to QKLMS. A high throughput pipelined implementation of KNLMS is described in [23] achieved by time multiplexing multiple problems. For a single problem or dataset the throughput of this architecture is much higher. Another approach to implement KNLMS algorithm is allowing for delayed updates to manage the dependency achieving a high throughput [22]. Braiding relies on properties of NORMA that are present in other algorithms such as [49] and hence could be applied there as well.

3.3 Summary

This chapter proposed a hardware architecture technique known as *braiding*. In this technique, partial results are computed at each stage of a pipeline and combined when possible. This technique was applied to NORMA. In addition the precision required for

NORMA to function effectively was also examined. Using these results, a very high throughput and low latency FPGA implementation was created.

Chapter 4

Neural Networks in Hardware

The computational complexity of CNNs imposes limits on certain applications in practice [36]. There are many approaches to this problem with a common strategy for the inference problem being to reduce the precision of arithmetic operations or to increase sparsity [7, 15, 18, 57, 71, 92]. It has been shown that low precision networks can achieve comparable performance to their full precision counterparts [15, 48, 92].

This chapter explores accelerating CNN inference with FPGA implementations of neural networks in fixed point arithmetic with ternary weight values. As the weights are restricted to $\{-1, 0, 1\}$, the multiplications between the inputs and the weights are reduced in complexity to subtractions, additions or no operation. Compared with previous work, the datapath is customised based on known weight values of a trained network at design time.

The computationally intensive parts of a CNN are matrix-vector multiplications. Given prior knowledge of the ternary weight matrix, this chapter shows matrix vector multiplications can be implemented as the evaluation of an unrolled and pruned adder tree, effectively storing the weights in the routing. This technique is only recently feasible due to increasing sizes of FPGAs, improvements in the tools and advances in low precision machine learning. The main contributions of this chapter are as follows:

- A novel architecture for inference of ternary neural networks on an FPGA by employing complexity reduction to the evaluation of pruned adder trees. In particular optimisations are: (1) streaming inputs and full pipelining of the computation; (2) ternary weights with 16-bit sums and activations to

preserve accuracy; (3) bit-serial summation for compactness; (4) weight-specific circuit generation with removal of multiplies by 0 and merging of common subexpressions to minimise computation; (5) throughput matching of CNN layers to minimise resource usage.

- An extension of a technique to train ternary neural networks described by Li et al. [48] allowing simple control of the sparsity of the network.

The work in this chapter was published in TRETS as *Unrolling Ternary Neural Networks* [79].

4.1 Previous Work On Hardware Acceleration Of Neural Networks

Previous work [6, 13, 24, 29, 36, 38, 50, 51, 58, 60, 67, 68, 83, 85, 90] has mainly focused on general accelerators, implemented either as a sequence of instructions on fixed hardware, or accelerator platforms designed for linear algebra intensive computation. Systolic array based architectures implement a grid of local-connected processing units to perform a matrix-to-matrix multiplication. Most notably, Jouppi et al. [36] describe the Tensor Processing Unit (TPU), an Application Specific Integrated Circuit (ASIC) that utilises a systolic array to compute operations necessary with 8 or 16 bit weights and activations. It provides a very high throughput on a variety of applications with a latency on the scale of 10ms, claiming a peak throughput of 92 TOps/sec and a sustained throughput of 86 TOps/sec. Similarly, Moss et al. [60] implemented a systolic array that utilised the Intel QuickAssist heterogeneous system with a CPU and FPGA, accelerating CNNs with binary activations and weights on ImageNet and achieving 40.96 TOps/sec. Venkatesh et al. [85] use a method described in [48] to implement a VGG style network with ternary weights and half-precision floating-point activations. They create an ASIC accelerator for training networks in addition to inference with a systolic array like structure. Due to the use of floating-point activations, they achieve high accuracy on CIFAR10 of around 91%.

Differing from the systolic array approach, vector processors contain several independent processing lanes, with the capability of each determined by the lanes' architecture. Chen et al. [13] created a custom ASIC utilising 16 bit fixed point achieving a peak throughput of up to 42 GMAC/s. Their architecture contains an array of independent processing elements, each receiving operation instructions. In the programmable logic domain, Wang et al. [29], Qiu et al. [68] and Meloni et al. [58] presented neural network accelerators for the Zynq CPU+FPGA. Each implemented a vector processor and utilised low precision to improve computational performance for object detection and image recognition. Wang et al. [29], Qiu et al. [68] and Meloni et al. [58] achieved 2 TOps/sec, 187 GOps/sec and 169 GOps/sec respectively.

Finally, Zhang and Prasanna [90] implemented an accelerator using the frequency domain representation of the convolution. Their datapath converts the activations into the frequency domain and uses a point-wise multiplication to perform the convolution; this is followed by an inverse FFT. The computation is performed in floating-point and implemented on the Intel QuickAssist platform containing a CPU and Stratix V FPGA, achieving 123.5 GFLOPs.

The following sections of this chapter will explore the computation required in convolutional neural networks, with a focus on optimizations for the case of ternary weights. Following the theory and proposed FPGA architecture, Chapter 5 will present two case studies for CIFAR10 image classification on the AWS F1 instance and automatic modulation classification on the RFSoC board from Xilinx.

4.2 Convolutional Neural Networks

Chapter 2 introduced the structure of Convolutional Neural Networks (CNNs). Of particular interest is equation 2.18 reiterated here as

$$\mathbf{y}_{i,j} = \mathbf{X}_{i,j}\mathbf{w} \tag{4.1}$$

This equation expressed the dependencies required for the computation of $\mathbf{y}_{i,j}$. When a network is trained and just being used for inference, $\mathbf{w}$ is known. The dependency for $\mathbf{y_{i,j}}$ arises from $\mathbf{X}_{i,j}$ which is a portion of the entire input. The critical point for convolutions in regards to latency is that only a portion of the entire input is required to produce an output. This operation is still a vector-matrix multiplication which requires multiplications followed by an accumulation of n results. The depth of the tree and hence the latency depends on the number of operations.

Introducing sparsity into $\mathbf{w}$ can decrease the number of operations further. Exploiting this sparsity, however, can still be very difficult in traditional computer architectures. The reason for this is the inefficiencies involved in moving data around to reach the processor. With sparse operations, the data required is typically stored in various different locations meaning alot of different blocks are loaded. There have been many attempts to deal with this, but it still presents issues.

In this work, the aim is to provide a very low latency implementation, while pipelining it to a reasonable degree to achieve a high throughput as well. In addition, a lower precision implementation is chosen which does have some accuracy cost. Lower accuracy may make this methodology unappealing for certain applications. However, some applications may be infeasible or impractical unless low enough latency or high enough throughput can be achieved.

4.2.1 CNN Components

This section describes the different components of a CNN with regard to a low latency and high throughput hardware implementation. The input and output of a hardware block can significantly constrain the possibilities. Creating a hardware block under the assumption that the whole input is available in memory can simplify the design. However, there is a latency associated with loading it into memory in the first place. Waiting for all inputs to be available before starting the computation adds an unnecessary latency

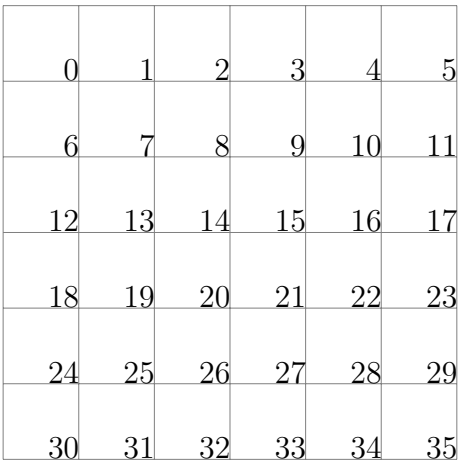


Figure 4.1: A small example image

overhead. To obtain data on a hardware device typically requires the serialization of the input as a very wide bus is typically not practical.

Hence, the scheme is adopted where the input arrives sequentially and is immediately available for computation. To obtain $\mathbf{X}_{i,j}$ from equation 4.1 requires a transformation. In the case of image inputs, this transformation is known as *im2col* or *im2row*. This transformation copies a single window of the input matrix into a much larger matrix where each input row is used to compute a single output pixel. For example, Figure 4.1 shows a small 6×6 image. The transformation of *im2row* for a 3×3 convolution without padding would give the inputs as shown in Table 4.1. The values are copied across so that each row represents all the inputs to a single convolution. The hardware block to implement this transformation will be known as the *Buffering* block.

Table 4.1: An example of the im2row transformation

0	1	2	6	7	8	12	13	14
1	2	3	7	8	9	13	14	15
2	3	4	8	9	10	14	15	16
3	4	5	9	10	11	15	16	17
6	7	8	12	13	14	18	19	20
...	...	...	...	...	...	...	...	...
21	22	23	27	28	29	33	34	35

Max Pool is a logical operation that is widely used in CNNs to reduce the computation needed. In terms of computational requirements, max pool is very simple. It only

requires determining the maximum value in a very small set of inputs. With reference to Figure 4.1, and assuming a 2×2 kernel the first output of the max pool is to compute $\max(0, 1, 6, 7)$. Similar to an accumulation, this would require $n - 1$ comparisons for n inputs and can be performed as a tree to reduce the latency.

To improve training times and reduce overfitting, batch normalization [33] is frequently used. During training there are four vectors that batch normalization uses to perform the update step on the inputs. These are the mean, variance and two learnable parameters γ and β sometimes called the scale and shift respectively. The mean and variance are just computed over a single batch during training. The population mean (μ) and variance (σ^2) are also collected during training and these values are used instead for inference. In inference, however, batch normalization simplifies to the transformation of $y(x) = ax + b$ where a and b are defined in Equation 4.2. ϵ is a small number to stop division by zero.

$$\begin{aligned} a &= \frac{\gamma}{\sqrt{\sigma^2+\epsilon}} \\ b &= \beta - a\mu \end{aligned} \tag{4.2}$$

Implementing $y(x) = ax + b$ in hardware is straightforward, requiring a multiplication with the input x followed by an addition. This operation requires few resources and only adds minimal latency.

The largest cost in terms of latency in CNNs is fully connected or dense layers. Unlike the convolution operation, a dense layer requires all the inputs to be used in the computation of each output. This means all previous layers in the network have to be complete before the dense layer can output a single result. Fortunately, useful work can still be done to compute partial results in the dense layer without the full set of inputs.

The hardware implementation of logical blocks in the CNN are described in the following sections

- Buffering - (im2col/im2row)
- Max Pool
- Convolution

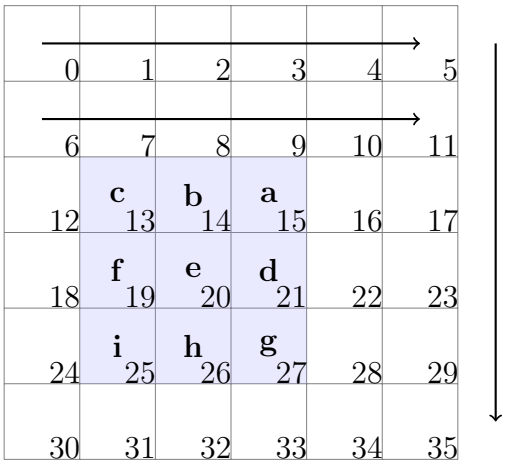


Figure 4.2: Streaming the Image and the Window to Buffer

- Scale and Shift - (Batch Normalization Inference)
- Dense Layers

4.2.2 Buffering

There are two main approaches in the literature to compute the convolution in hardware: passing in inputs over multiple cycles and storing intermediate results when computing the convolution; or buffering the pixels so that the entire set of inputs needed for the convolution is available simultaneously such as the approach in Baskin et al. [6] and Prost-Boucle et al. [67]. This work explores the scheme used by these authors to buffer the pixels, so the entire set of inputs is available simultaneously. It buffers previous inputs in such a way that each cycle it can output the current vector $\mathbf{X}_{i,j}$ from Equation 4.1. The inputs necessary to compute an output pixel of the convolution are only a small segment of the image. As the image is streamed into a layer of the CNN, it is buffered in order to transform the pixels into a patch of the image from which a convolution or Max Pool operation can be computed. This is referred to as a buffering block in this thesis but is also known as the im2row or im2col transformation. The purpose of the buffering hardware block is to stream the image through a fully pipelined design with p pixels provided each cycle as input from a FIFO. To compute an output pixel of the convolution requires using inputs from various pixels of the image.

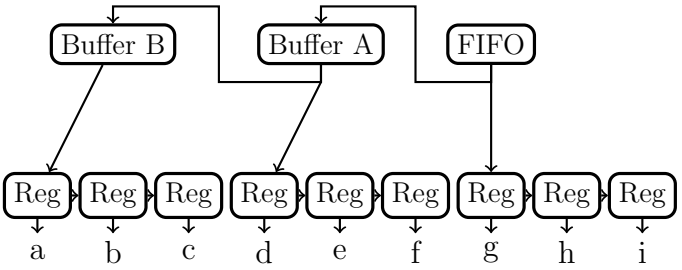


Figure 4.3: Diagram illustrating buffering for a 3×3 convolution

Figure 4.2 demonstrates the input configuration used in this work with the assumptions that the convolutional kernel is 3×3, the image is $6 \times 6 \times 1$ and a single pixel is streamed in each cycle. The image is streamed in left to right, top to bottom with a pixel arriving in the cycle indicated by the number in each box. The input *FIFO* in Figure 4.3 outputs the pixel each cycle to both *Buffer A* and the first stage of a shift register. The value is then shifted across each cycle by the registers as the window moves across the image. *Buffer A* and *Buffer B* delay the output by the image width (6 cycles in Figure 4.2) in order to output the previous two rows of the image with matching columns. For example, when the *FIFO* outputs pixel 27 in Figure 4.2, *Buffer A* outputs pixel 21 and *Buffer B* outputs pixel 15. After the outputs *a*-*i* are obtained at the bottom of Figure 4.3, there is additional logic necessary to implement zero padding by switching between these values and zero. This produces the correct vector $\mathbf{X}_{i,j}$ from Equation 2.21 for the convolution. This can also be adapted for *Max Pool* layers where the structure of Figure 4.3 changes depending on the kernel size, image size and pixel rate p.

4.2.3 Max Pool

Max pool layers are widely used in CNNs to downsample the image. The max pool operation takes a $k \times k$ window of the image as input similar to that shown in Figure 4.2. For these pixels, it compares them and outputs the maximum value resulting in a single pixel. Reduction in the image size is achieved by using a stride greater than $n = 1$. Consider an example based on the image in Figure 4.2 with a kernel size of $k = 2$:

if the stride is only $n = 1$, the window of pixels $0, 1, 6, 7$ would be followed by pixels $1, 2, 7, 8$ meaning the output image is about the same size. To downsample the image, it is common to use a stride of $n = 2$ or more in max pool layers. With a stride of $n = 2$, the window of pixels $0, 1, 6, 7$ is followed by pixels $2, 3, 8, 9$ which would produce a 3×3 image from the input in Figure 4.2. The presence of max pool layers has the effect of reducing the amount of computation required for subsequent layers of the CNN. The downsampling property of max pool is exploited in the proposed architecture to reduce hardware area. Given a throughput of p pixels/cycle as input into a max pool layer with a stride of n results in the output throughput reduced to $\frac{p}{n^2}$ pixels/cycle. This property is later used to dramatically reduce the size of the convolutional layers in hardware through the use of word and bit serial adders discussed in Section 4.2.4.

As a hardware block, the max pool is implemented in a pipelined way comparing the k^2 inputs over multiple cycles to determine the output for each component of the pixel. When using the input configuration from Figure 4.2, the output of the max pool is bursty, either producing results frequently on particular image rows or otherwise not generating any output. This requires the use of a *FIFO* at the output of a max pool layer. When using a stride of 2, this bursty nature is caused by the output occurring every 2nd row for a 2D convolution. In the case of a 1D convolution, the output is not bursty and hence does not require a FIFO on the output.

4.2.4 Convolution

Convolutional layers are where the bulk of the computation is performed in the network and hence require a large portion of the total hardware. Reducing the size of this hardware is essential to the viability of computing a CNN with low latency. Most machine learning implementations utilize floating-point numbers to represent a large dynamic range. Unfortunately, floating-point is expensive in hardware [31].

As discussed in Chapter 2, significant recent research has focused on low precision neural networks. Lower precision can dramatically decrease the size of hardware required to

compute the operations. This comes at the cost of accuracy of the CNN. Finding the right balance between precision of the calculation and the resultant accuracy depends on the application.

For this work, ternary weights are chosen for the convolution operation. Restricting the weights to $-1, 0, 1$ has several advantages in hardware. The most significant of these is that multiplications between the activations and weights no longer need to be performed. Rather, it is now a choice between adding, ignoring or subtracting an activation. The other benefit of using this precision is the inclusion of 0 in the set of available weights. This introduces sparsity into the calculation, reducing the number of operations. If exploited properly, this can significantly reduce the computation required.

To fully exploit this, it is assumed that the convolution uses ternary weights known and fixed at design time. The im2col transformation described in Section 4.2.2 produces a dense vector to multiply into the convolutional weights matrix. To multiply this into the convolutional weights, consider the example convolutional weights shown in Table 4.2. This shows a single trinarised 3×3 convolutional filter on a grayscale image centered on an input pixel with value 'e'. With reference to equation 4.1, $\mathbf{X}_{i,j}$ is represented by the variables $a, b, c, d, e, f, g, h, i$. These are the inputs to the convolution and depend on the input image used and hence are unknown at design time. It is assumed that the weights $\mathbf{w}$ in equation 4.1 are known ahead of time. These known weights are multiplied with the inputs as shown in table 4.2. The results of these multiplications need to be accumulated. The output of this convolution operation is then given as $z_0 = -a + c + e + f - h$. It is important to note that this equation calculating z_0 can be implemented without multiplications hardware and allows the removal of zero weights. It is also interesting to note that, if implemented as an adder tree multiplying in the weights and using all 9 inputs would initially require a depth of 4. In this reduced form, the adder tree only needs to have a depth of 3 meaning a lower latency for the computation.

More generally, the convolution computes Equation 2.21 which is a sparse matrix vector operation where the matrix is ternary and known at design time. The sparse matrix

Table 4.2: An example 3×3 Convolutional Filter

a $\times$ (−1)	b $\times$ 0	c $\times$ 1
d $\times$ 0	e $\times$ 1	f $\times$ 1
g $\times$ 0	h $\times$ (−1)	i $\times$ 0

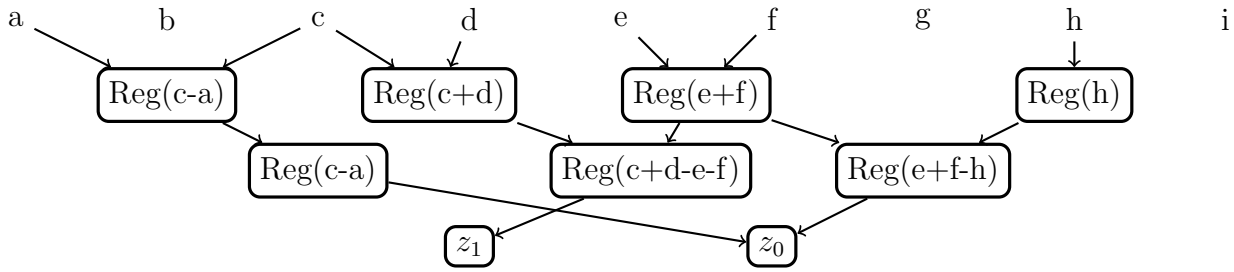


Figure 4.4: Computing $z_0 = c + e + f - (a + h)$ and $z_1 = c + d - e - f$

vector multiplication can be implemented as a row of multiplications followed by a different summation for each output. The implementation of the convolution in this thesis aims to exploit sparsity by removing any multiplications with zero, reducing the size or pruning the adder tree required to compute the summation. The remaining values are multiplied by either -1 or 1. Hence the multiplications can be removed entirely from the design with the 1 or -1 weight just indicating whether to add or subtract from the sum.

As there are typically numerous filters in the convolution, many summations are performed in parallel. Using knowledge of the weights ahead of time, it is possible to merge subexpressions within the summations to reduce hardware usage. An example of how knowledge of the weights ahead of time can reduce the amount of hardware through subexpression elimination is shown in Figure 4.4. Here, an additional equation $z_1 = c + d - e - f$ is computed in parallel to z_0 simultaneously and sharing the subexpression $e + f$.

4.2.5 Scale and Shift

Batch Normalization (BN) is included in nearly every network due to its ability to improve training times and reduce overfitting. In feed-forward mode, it is reduced to a

simple equation

$$\mathbf{y} = \mathbf{a} \odot \mathbf{x} + \mathbf{b} \tag{4.3}$$

where $\mathbf{x}$ represents activations, $\mathbf{a}$ and $\mathbf{b}$ are constants to scale and shift from the batch normalization operation known at design time and $\odot$ is the elementwise product. Equation 4.3 is referred to in this thesis as a scale and shift operation. As BN typically directly follows the convolutional layer, the scaling coefficient of the weights s from Equation 2.21 can be combined into a single equation

$$\mathbf{y} = \mathbf{c} \odot \mathbf{x} + \mathbf{b} \tag{4.4}$$

where $\mathbf{c} = s \cdot \mathbf{a}$ as $\mathbf{a}$ and s are constants known at design time. The bias from the convolutional layer above can also be integrated into these variables.

The Scale and Shift hardware block can be implemented with a multiplier and adder using the constant values of c, b for each output followed by an activation function. This requires a fixed point multiplication with a constant after each layer, for each output filter. This is achieved by having a multiplication operation at all outputs followed by an adder for the bias and finally having the activation function applied on y. As the multiplications are with constants, depending on their value the multipliers might be implemented with or without DSPs at the discretion of the tools.

4.2.6 Dense Layers

CNNs can include dense layers at the end of the network for the final classification. The output of the convolutional layers is flattened, then passed into the dense layer. Unlike the convolutional layers, the dense layer requires the entire image to be available before any of the outputs can be computed.

While the dense layer is also a matrix vector operation, it typically has a significantly larger number of weights, meaning it is expensive to expand it into a tree as with the convolutional layers. Additionally, as a dense layer is operated on the vector of the entire image, it would be inefficient to use the technique described for convolutional

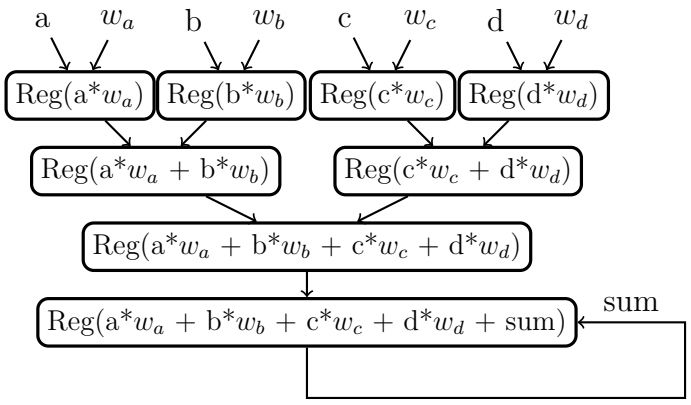


Figure 4.5: Multiplying and Accumulating Weights for the Dense Layer

layers. The main reason for this is the pruned adder tree approach would require the entire image to be buffered before beginning the computation. This means the hardware to compute the dense layer is only utilised once per image. This would be inefficient and would have an unreasonable hardware requirements. For this reason, taking advantage of sparsity and Common Subexpression Elimination (CSE) such as in Figure 4.4 is not efficient for dense layers. Hence, the most common approach in the literature is adopted which streams the weights from memory, multiplies them into the activations and accumulates the result as shown in Figure 4.5. As the weights are known ahead of time, they can be stored in read only memory blocks on chip. The number of Multiply Accumulate (MAC) units as shown in Figure 4.5, depends on the size of the dense layer. Each output of the dense layer has its own accumulation block.

4.3 Reducing Hardware Area

The hardware cost of implementing a fully pipelined CNN is high. Minimizing this is an essential element of a low latency CNN. Two main approaches are considered to reduce the hardware area. The first is a higher level approach trying to share partial results of the network to find and remove redundant computations. This is also referred to as subexpression elimination. Making use of idle time in an unrolled implementation to save on hardware is the second approach.

The CSE techniques discussed in this section are implemented in the pip3 package *twn_generator* developed in the course of this work. The package includes implementations of the CSE methods introduced in sections 4.3.3 and 4.3.4. Additionally, it includes the methods for generating Verilog with adders of varying precision to reduce hardware area.

4.3.1 Subexpression Elimination

This section discusses techniques that can be applied to merge common subexpressions in the implementation of the pruned adder trees to reduce the hardware required for implementation. The subexpression elimination problem has a long history in computer science and its solvers are widely used in many applications such as the GCC compiler. It has been determined to be NP-hard and hence can only be solved approximately for large problem sizes [9]. The subexpression elimination discussed in this dissertation is a specialized case of the problem. It is an unknown dense vector multiplied by a sparse matrix of constant values, restricted to $-1, 0, 1$. Figure 4.4 shows the desired solution to the problem given the input equations $z_0 = c + e + f - (a + h)$ and $z_1 = c + d - e - f$. The solution should be a pipelined adder tree aiming for the smallest implementation on an FPGA. As an FPGA has registers after adder blocks, the cost for Add+Reg is considered the same as just Reg and should optimize this for minimal cost. The particular implementation of the Add+Reg block is considered irrelevant. There are three related approaches to this problem in the literature. RPAG by Kumm et al. [44] use constant integer values in the matrix. Hsiao et al. [32] use Common Subexpression Elimination (CSE) on a binary matrix for implementation of AES in hardware which will be modified for ternary values and referred to as top-down CSE (TD-CSE) in this work. Wu et al. [89] expand this to gate level delay computing CSE (GLDC-CSE) for binary values which will be modified for ternary values and referred to as bottom-up CSE (BU-CSE) in this work.

The constant matrix optimization RPAG of Kumm et al. uses the graph-based algorithm [45] to solve the generalized constant multiplication problem where the weights can

be arbitrary integer numbers instead of restricted to $-1, 0, 1$. Conversely, Hsiao et al. requires values of either 0 or 1 to use TD-CSE and is hence a specialization of the problem required to be solved in this work.

4.3.2 RPAG Algorithm

Kumm et al. [44] propose a method to perform subexpression elimination with integer values. The algorithm looks at all the outputs of a matrix-vector multiplication and calculates the minimal tree depth, d, required to get the results. At this depth, it then tries to determine the minimum number of terms needed at depth $d-1$ to compute the terms at depth d. This is then performed iteratively until the depth is 1 resulting in the whole tree being generated. These steps perform a broad search of the space and hence find very good solutions. However, this method searches for common subexpressions allowing for more general integer coefficients. In this case, it is not necessary. This method and the broad search make the approach computationally intensive and only suitable for relatively small matrices.

4.3.3 Top-Down CSE Approach

The TD-CSE algorithm proposed by Hsiao et al. [32] iteratively eliminates subexpressions of size two. To explain the TD-CSE approach, consider the following example of a maxtrix-vector product:

$$\mathbf{y} = \begin{pmatrix} 0 & 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 1 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 0 & 0 & 1 \\ 1 & 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 1 \end{pmatrix} \begin{pmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \end{pmatrix} = \begin{pmatrix} x_2 + x_3 \\ x_0 + x_2 + x_3 + x_4 \\ x_1 + x_4 + x_5 \\ x_1 + x_5 \\ x_0 + x_2 + x_3 \\ x_0 + x_3 \\ x_1 + x_4 + x_5 \end{pmatrix}. \tag{4.5}$$

The TD-CSE algorithm counts the frequency of all subexpressions of size two, selecting the most frequent. In this case, the most common pair is $x_2 + x_3$, occurring three times. This expression is removed by defining $x_6 = x_2 + x_3$ and replacing all previous equations with x_6 resulting in

$$\mathbf{y} = \begin{pmatrix} x_6 \\ x_0 + x_4 + x_6 \\ x_1 + x_4 + x_5 \\ x_1 + x_5 \\ x_0 + x_6 \\ x_0 + x_3 \\ x_1 + x_4 + x_5 \end{pmatrix} . \tag{4.6}$$

This process continues until subexpressions do not occur more than once.

This approach can be thought of as building multiple adder trees from the inputs to the outputs by creating an adder each iteration. In this case, the adder $x_2 + x_3 = x_6$ was put into the tree. There are some tricks to implement this approach that can dramatically improve the runtime. When calculating the most common pairs, the results can be stored and reused as, after removing a subexpression, most of these values do not need to be recalculated. In the example in equation 4.5, the number of times each pair occurs has to be computed to recognize that $x_2 + x_3$ is the most frequent. After this has been determined, the frequency of pairs such as $x_4 + x_5$ can just be stored. Some expressions do need to be checked and updated such as $x_0 + x_3$. As the expression $x_2 + x_3$ was removed from $x_0 + x_2 + x_3$ and $x_0 + x_2 + x_3 + x_4$, the frequency of the expression $x_0 + x_3$ needs to be reduced by two. However, most of the time, the frequencies of the pairs do not need to be recalculated such as $x_4 + x_5$.

Expressing this more formally, the first iteration requires the number of expressions containing each pair of variables to be computed where n is the number of expressions, v is the number of variables and p is the number of pairs of variables in the expressions.

For example, in Equation 4.5, $n = 7$, $v = 6$ and $p = \binom{v}{2} = \frac{v(v-1)}{2}$. The complexity of the first iteration to find the frequency of all pairs is then $O(nv^2)$. These results are then stored. When a pair of variables are chosen to be removed as a common subexpression, the computation of the update is only $O(nv)$. For the example in Equation 4.5, after the subexpression is removed, only combinations containing x_2, x_3 and x_6 would need to be computed. The count of previous pairings between say, x_0 and x_1, do not need to be updated. Another minor optimization to further skip computations is to look at combinations with x_2 and x_3. If there were no expressions in common in the initial equations, say between x_2 and x_5, then the initial calculations would have a value of zero here. As the number of subexpressions can only decrease after removing a subexpression, this value of zero does not need to be recomputed and the pair between x_2 and x_5 can be skipped. As the problem grows in size, the amount this saves increases but the update is still $O(nv)$. It should also be noted that v grows by 1 each new subexpression that is removed as a new variable is created for it. This method is generalized to $-1, 0, 1$ reasonably easily by creating another table for subexpressions with different signs.

4.3.4 Bottom-Up CSE Approach

Wu et al. [89] propose a method they term Gate-Level Delay Computing CSE (GLDC-CSE) for AES S-Box implementation. We expand this method to $-1, 0, 1$ instead of $0, 1$ and refer to it as bottom-up CSE (BU-CSE).

GLDC-CSE considers building the tree from the opposite direction to TD-CSE similar to RPAG. Instead of starting at the inputs, it starts at the outputs and works back to the inputs. Compared with TD-CSE, finding common expressions is more computationally intensive but can find better results as larger common subexpressions are preferred. When building the tree from the bottom up, the size of the largest common subexpression needs to be determined for every pair of vectors. The largest common subexpression is then selected to be removed. For the example in Equation 4.5, $x_0 + x_2 + x_3$ appears twice. The subexpression, $x_6 = x_0 + x_2 + x_3$ is added to the table of updated equations

leading to

$$\mathbf{y} = \begin{pmatrix} x_2 + x_3 \\ x_4 + x_6 \\ x_1 + x_4 + x_5 \\ x_1 + x_5 \\ x_6 \\ x_0 + x_3 \\ x_1 + x_4 + x_5 \\ x_0 + x_2 + x_3 \end{pmatrix}. \tag{4.7}$$

It is important to note that there is a new row of $x_0 + x_2 + x_3$ at the bottom added to the equations. This is because inside of x_6, there are still other subexpressions that can be combined. For example, the matrix still contains a common subexpression of $x_2 + x_3$ or $x_0 + x_3$ that can be removed.

The algorithm for removing common terms is described as follows:

(1) Compute the number of common terms for each pair of vectors and store this as the *pattern matrix*
(2) Find the largest value in the pattern matrix and the vectors to which it corresponds
(3) Remove that subexpression from all matching vectors following the process described for the example in Equation 4.7
(4) Update the *pattern matrix*
(5) Go to step 2 until the largest value in the *pattern matrix* is 1

To expand this to weights of $-1, 0, 1$ the computation of the pattern matrix becomes slightly more complicated. However, the methodology remains the same.

4.3.5 Opportunities for further improvement

Due to the scale of the problem, an exhaustive search of the space for CSE is not feasible. RPAG does a much broader search than the greedy bottom up or top down CSE approaches. One possible path for further work could be to find a way to reduce the search space of RPAG and find a way to make it massively parallel. The top down and bottom up CSE methods are already searching a minimal portion of the space. These CSE algorithms could be adapted to randomly choose a slightly less optimal path occasionally. Perhaps with many runs, a better solution could then be found.

4.3.6 Exploiting Idle Hardware

The computation required to compute each layer in a CNN can vary significantly. This property of the network leads to an unequal distribution of computational load amongst the network. This is caused by reduction operations, such as max pooling.

When a fully pipelined adder tree is used to implement the convolutions as discussed in section 4.2.4, these reduction operations can lead to the tree being idle for multiple cycles. When the number of inputs each cycle is ≥ 1, it is straightforward to construct an efficient hardware block by implementing multiple versions of the adder trees in parallel. This hardware will be utilized every cycle. After reduction operations, the number of inputs each cycle can decrease resulting in idle cycles and an inefficient design. Time multiplexing the computation for different sets of weights destroys a lot of the advantages of the adder tree approach as it becomes much more difficult to eliminate subexpressions and exploit sparsity.

There are two approaches that can take advantage of this idle time and remove this inefficiency in the implementation. The first is to compute the additions over multiple cycles. This is desirable when the inputs into the convolution have a sufficiently high bitwidth. The second approach is to increase the precision in the adder tree so that a higher accuracy can be obtained. This is useful when very low precision inputs are being used.

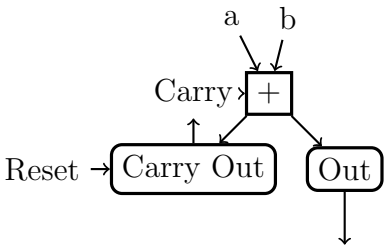


Figure 4.6: Word or Bit Serial Adders/Subtractors

4.3.6.1 Computing over multiple cycles

When computing over multiple cycles the bitwidth used is important. In this case, 16 bit inputs are assumed. A 16 bit adder requires 2 slices on a Xilinx Ultrascale FPGA to be implemented. If this computation is performed over 4 cycles with a 4 bit word size, the area required to implement the adder is only half a slice. With a bit serial adder, the result is calculated over 16 cycles and only requires a single LUT or $\frac{1}{8}$ of a slice.

Figure 4.6 shows a diagram of how addition or subtraction can be computed over multiple cycles. The reset discards carry information to begin computation of the next set of numbers. For addition, the carry is reset to 0. For subtraction with 2s complement, the carry is reset to 1. The inputs a and b are added with the carry to output the result for that word or bit. In the case of subtraction, b will need to be inverted bitwise which is implemented inside the same LUT. The carry out is stored for the next word or bit to be computed. The output is passed directly into the next adder of the tree.

4.3.6.2 Increasing Precision

Another way to exploit idle cycles in the adder trees is to use higher precision. In a similar way to the bit or word serial adder, extra cycles can be utilized if the precision is increased. Consider, for example, if binary inputs were used in the adder tree. The adder tree is now implementing a pop-count on these binary inputs. If the adder tree is only needed every two cycles, increasing the precision of the inputs to 2 bits exploits idle time. The adder tree is used twice over two cycles to compute the result.

Increasing the precision in this way does not impact the hardware used significantly but can provide a simple way to improve the accuracy of the model.

4.4 Summary

This chapter explored the implementation of TNNs for inference under the assumption that the weights are known. It described how to implement different components of a TNN in hardware for an FPGA. Of particular note is the implementation of convolutional layers. Examining the convolution operation, it was determined that convolutional layers can be implemented without multipliers if the weights are constrained to -s,0 and s. If an irregular adder tree is used to compute the convolution operation, sparsity is very effectively exploited. Additionally, CSE can be applied to this adder tree to further reduce computational requirements. The next chapter will explore two case studies for the application of this technique.

Chapter 5

Unrolling Case Studies

Chapter 4 proposed methods to implement components of CNNs in hardware. It particularly focused on the case where the weights of the network are restricted to $-1, 0, 1$ and known at design time. This led to the proposal of a technique of unrolling the network and using CSE to reduce hardware utilization. This chapter focuses on the application of this to two different problems using different systems. The first is the well known dataset, CIFAR10. The task in this dataset is to classify relatively small images (32×32) into ten classes. This is implemented on an AWS F1 instance and was published in TRETS as *Unrolling Ternary Neural Networks* [79].

The second is for the problem domain of Automatic Modulation Classification (AMC) where the type of modulation in Radio Frequency (RF) transmissions needs to be determined. Detecting what modulation is used can give insights into the device used to transmit it and the capabilities of the RF channel. Recently O'Shea et. al. [62] published a dataset with 24 different modulation classes. Section 5.2 implements a CNN in hardware to classify these signals on the Radio Frequency System on Chip (RFSoC) board also known as the ZCU111. A full system implementation is created, reading inputs from the ADC and classifying them. Section 5.2 was accepted to FPT-19 as *Real-time Automatic Modulation Classification* [78]. Section 5.2 also includes some extensions beyond this paper to lower precision activations.

Both implementations achieve very low latency and high throughput compared with published literature. The CIFAR10 dataset and O'Shea datasets were chosen to explore the unrolling technique. This allows an exploration of both 1D and 2D convolutions. The more complex dataset, imagenet, was not chosen as it requires very large networks

and would have required multiple FPGAs. As this work is a demonstration of the technique rather than a solution for a particular application, CIFAR10 is sufficient. As the networks contain a variety of layer sizes, they effectively evaluate the methodology for implementing convolutional layers.

5.1 CIFAR10 Case Study

This section considers applying the techniques discussed in Chapter 4 to the network described in Li et al. [48], on the CIFAR10 dataset which provides 10 different image types: airplane, automobile, bird, cat, deer, dog, frog, horse, ship and truck. A CNN similar to VGG-7 was trained using a scheme adopted from Li et al. The following parameters were used:

- A batch size of 128
- An initial learning rate of 0.1
- Learning rate decays by a factor of 0.1 every 100 epochs.
- Run for 120k steps or 307.2 epochs
- Adam optimizer was used

The models were also trained with data augmentation as described by Li et al. [48] where the image is padded with 4 pixels to each side and randomly cropped back to the 32×32 image size during training. Li et al.'s method of training a low precision CNN utilises a threshold

$$\Delta^* \approx \epsilon \cdot E(|W|) \tag{5.1}$$

where Li recommend a value of $\epsilon = 0.7$. If the magnitude of the floating-point weight is less than this threshold, Δ^*, it is set to 0. Otherwise the sign of the floating-point weight is used multiplied with a scaling factor s. The parameter ϵ thus directly controls the sparsity of a layer in this method.

This technique of adjusting ϵ to trade off accuracy and sparsity of the network simply and quickly is not explored in Li et al. [48] but is a strong feature of their method and

Table 5.1: Effect of ϵ on sparsity and accuracy for CIFAR10

TNN Type	ϵ	Sparsity (%)	Accuracy (Top 1)
Graham [26] (Floating Point)	-	-	96.53%
Li et al. [48], full-size	0.7	≈ 48	93.1%
Half-size	0.7	≈ 47	91.4%
Half-size	0.8	≈ 52	91.9%
Half-size	1.0	≈ 61	91.7%
Half-size	1.2	≈ 69	91.9%
Half-size	1.4	≈ 76	90.9%
Half-size	1.6	≈ 82	90.3%
Half-size	1.8	≈ 87	90.6%

important to this approach as sparsity directly effects the size of the hardware. The network architecture proposed by Li et al. was modified by reducing the number of filters by half in all convolutional layers of the network and reducing the size of the dense layer eight fold before training. This was done as the impact on accuracy is minor but the size of the network for a hardware implementation is dramatically reduced.

The results obtained for different values of ϵ are summarised in Table 5.1. Note that the result of the implementation in the second row of the table is higher than the 92.6% reported by Li et al. [48] which is likely due to longer training time. As Table 5.1 shows, the choice of $\epsilon = 0.7$ produces a network with high accuracy but is relatively dense, with a sparsity of around 47%. The remaining results were obtained by maintaining $\epsilon = 0.7$ in the first layer, $\epsilon = 1.0$ for the dense layers and adjusting all the other convolutional layers to use a larger value of ϵ. As ϵ increases, a slight drop in accuracy is observed with an increase in sparsity dependent on the value shown in Table 5.1.

With the value of $\epsilon = 1.4$ chosen from Table 5.1, the computation required for the network is reduced by almost half compared to a value of $\epsilon = 0.7$. This is extremely advantageous as, for this works implementation of Equation 2.21, increased sparsity reduces the area allowing a larger design to fit on a FPGA. The final network architecture chosen is given in Table 5.2. Batch normalization and ReLU activations are used after each convolutional and dense layer.

Table 5.2: Architecture of the CNN used for CIFAR10

Layer Type	Input Image Size	Num Filters	ϵ	Sparsity
Conv2D	32 x 32 x 3	64	0.7	54.7%
Conv2D	32 x 32 x 64	64	1.4	76.9%
Max Pool	32 x 32 x 64	64	-	-
Conv2D	16 x 16 x 64	128	1.4	76.1%
Conv2D	16 x 16 x 128	128	1.4	75.3%
Max Pool	16 x 16 x 128	128	-	-
Conv2D	8 x 8 x 128	256	1.4	75.8%
Conv2D	8 x 8 x 256	256	1.4	75.4%
Max Pool	8 x 8 x 256	256	-	-
Dense	4096	128	1.0	76.2%
Softmax	128	10	1.0	58.4%

Table 5.3: Comparison of CSE techniques on ternary weights with 2-input adders (top) and 3-input adders (bottom) for the first layer

Technique	Avg Adders	Avg Reg	Avg Add/Reg	Avg Time (s)
Baseline (2-input)	755.2	146.5	901.7	-
RPAG (2-input)	460.6	26.7	**487.3**	52.3
TD-CSE (2-input)	300.4	309.8	610.2	0.317
BU-CSE (2-input)	296.3	345.7	642	0.459
Baseline (3-input)	406.2	54.1	460.3	-
RPAG (3-input)	324.5	7.5	**332**	28460.8
TD-CSE (3-input)	246.7	294.5	541.2	0.297
BU-CSE (3-input)	258.8	321.7	580.5	0.474

5.1.1 The Impact of Subexpression Elimination Techniques

To compare the effectiveness of the three CSE techniques described in Section 4.3.1 ten models with the above network, each with a random initialisation, were trained with a chosen $\epsilon = 1.4$. The accuracy obtained from these models with floating-point activations is $91.7\% \pm 0.1\%$ and using 16 bit fixed point activations is $90.9\% \pm 0.1\%$. The first layer was selected from all ten trained models with a matrix size of 27 inputs and 64 outputs where all values are $-1, 0, 1$. The average results of the subexpression elimination techniques are compared in Table 5.3 for 2-input as well as 3-input adders.

It can be seen RPAG has the lowest register and adder combined cost. This is likely the smallest hardware cost for an FPGA due to the structure of adders having optional

Table 5.4: Comparison of CSE techniques on ternary weights with 2-input adders (top) and 3-input adders (bottom) for the second layer

Technique	Avg Adders	Avg Reg	Avg Add/Reg	Avg Time (s)
Baseline (2-input)	8936.4	254.3	9190.7	-
TD-CSE (2-input)	3970.6	1521.5	5492.1	**20.68**
BU-CSE (2-input)	3890.3	902.0	4792.3	55.94
Baseline (3-input)	4536.7	91.6	4628.3	-
TD-CSE (3-input)	2826.8	1447.9	4274.7	20.78
BU-CSE (3-input)	2425.6	439	**2864.6**	58.59

registers at the outputs. The cost on the FPGA of an adder and a register compared with a register alone is similar, hence the rightmost column provides the most meaningful comparison [45]. These results suggest that RPAG is the best technique, followed by TD-CSE, then BU-CSE.

However, experiments showed that RPAG does not scale well to the larger layers as shown by the significant running time for the smallest layer in Table 5.3. The scalability of these techniques to larger matrices is significant for this application, as the matrix size grows significantly in the subsequent layers. For those, the TD-CSE method is the most scalable as it can skip large portions of the computation each cycle. These results only reflect the first layer of a network with more outputs than inputs. In all subsequent layers there are more inputs than outputs in the rest of a CNN.

Table 5.4 shows a comparison between the TD-CSE and BU-CSE techniques for the larger matrices in layer 2. These matrices had 576 inputs and 64 outputs with a sparsity of around 75%. These results show that the BU-CSE technique finds better solutions for this configuration. It was not possible to run RPAG on these matrices as the large size makes it infeasible. This suggests that BU-CSE is better for larger matrices. The results, however, are similar enough that both should be run to find a better solution.

In the following, one out of the ten models trained for $\epsilon = 1.4$ was selected and applied CSE to its weights in all convolutional layers to reduce the amount of computation required for the FPGA implementation. Table 5.5 shows the results of running subexpression elimination on the convolutional layers of the trained network described in

Table 5.5: Common Subexpression Elimination on the Convolutional Layers; Only 2-input adds are compared in this table for consistency

Layer	Method	Adds	Regs	Adds+Regs	Time(s)	Mem(GB)
1	Baseline	731	137	868	-	-
	RPAG	451	31	**482**	64	0.008
	TD-CSE	295	304	599	0.4	0.029
	BU-CSE	295	321	616	0.5	0.03
2	Baseline	8432	249	8681	-	-
	TD-CSE	3782	1517	5299	24	0.1
	BU-CSE	3686	858	**4544**	64	0.17
3	Baseline	17481	491	17972	-	-
	TD-CSE	8466	2299	10765	89	0.18
	BU-CSE	8492	1878	**10370**	545	1.1
4	Baseline	36155	586	36741	-	-
	TD-CSE	17143	4214	21357	873	0.63
	BU-CSE	17309	3056	**20365**	2937	6.6
5	Baseline	71050	1198	72248	-	-
	TD-CSE	32829	6830	39659	3088	1.2
	BU-CSE	33026	6109	**39135**	25634	44
6	Baseline	144813	1270	146083	-	-
	TD-CSE	62653	13852	76505	26720	4.8
	BU-CSE	63832	10103	**73935**	147390	191.0

Table 5.2, comparing different techniques. The table does not show RPAG for layers 2-6 as it becomes computationally prohibitive. The reason that only the first two layers use 3-input adders in Table 5.2 is that the remaining layers compute the result over multiple cycles as discussed in Section 5.1.2. Here, 3-input adders can not be efficiently utilized.

The results from Table 5.3 show that RPAG is the most successful on the first layer of the network. Table 5.5 shows that, except for the first layer, BU-CSE finds the superior solution on the chosen network. One possible reason is the relative numbers of inputs and outputs as shown by the contrasting results in Tables 5.3 and 5.4. Layer 1 matrices used for the comparison in Table 5.3 have a smaller number of inputs than outputs. Conversely, layers 2-6 have a much larger number of inputs than outputs. The scalability of BU-CSE and TD-CSE should be mentioned as the time taken to

Table 5.6: Hardware Usage of the Convolutional Layers

Layer	Method	CLB/148K	FF/2.4M	LUTS/1.2M	P&R(hrs)
1	Baseline	1400	8723	8272	0.5
	BU-CSE	820	4499	5230	0.45
2	Baseline	15231	119848	116345	1.08
	BU-CSE	10258	71908	66131	0.93
3	Baseline	15171	102657	77743	1.9
	BU-CSE	8772	61965	36611	1.13
4	Baseline	30536	206940	164458	4.25
	BU-CSE	16909	118476	73581	2.68
5	Baseline	18414	165794	85743	3.86
	BU-CSE	7579	89820	39805	1.72
6	Baseline	35117	335134	180402	11.15
	BU-CSE	13764	160634	74696	3.08

Table 5.7: Improvement in resource usage when applying BU-CSE vs Baseline

Layer	% in Adds+Regs	% in CLBs	%in FFs	% in LUTs
1	-29.0	-41.4	-48.4	-36.8
2	-47.7	-32.6	-40.0	-43.2
3	-42.3	-42.1	-39.6	-52.9
4	-44.6	-44.6	-42.3	-55.3
5	-45.8	-58.8	-45.8	-53.6
6	-49.4	-60.8	-52.1	-58.6

run BU-CSE on the 6th (largest) convolutional layer is over a day, where as TD-CSE only needs a couple of hours and significantly less memory. This time is relatively insignificant for a model updated reasonably infrequently and can be considered an extension of the training time.

The effectiveness of the CSE results can be compared by running Vivado with aggressive area optimizations in synthesis and implementation in out of context mode. The Verilog generated without CSE ('Baseline' in Table 5.6) is compared to the Verilog generated after optimization with the BU-CSE algorithm. The effectiveness of CSE is compared with the difference in the originally generated Verilog in columns 'Adds', 'Regs' and 'Adds+Regs'. The runtime and peak memory used to run the CSE algorithm is shown columns 'Time' and 'Mem'. The last four columns show the logic resources after Place

and Route (P&R) as well as its runtime. Table 5.7 compares the rows for Baseline and BU-CSE from Table 5.6 by calculating the decrease when using CSE. The results demonstrate that the solution to the CSE problem of an entire network layer is difficult for Vivado to manage. The reason for the difference in results may be the abstracted view of the problem that the CSE methods have. This allows them to explore the problem in more detail and find a better solution than Vivado. Due to the use of word and bit serial adders, the amount of area that is used for layers 3-6 is reduced significantly compared to the amount of adders required. As discussed previously in Section 4.2.6, it is ineffective to use the pruned adder tree for the dense layers and thus CSE is not used for these.

5.1.2 FPGA Implementation

The network described by Li et al. [48] contains floating-point activations which limit FPGA performance as it is hardware intensive to implement add and multiply blocks. After the network was trained, the batch normalization variables and scaling coefficients for the convolutional layers were extracted from the model. A python script then computed the network performance using these weights on the CIFAR10 test set in fixed point. By outputting the maximum absolute values obtained at various stages of the calculation, a decision was made to use a total of 16 bits to maintain accuracy, ensure no overflow and to simplify the implementation of the word and bit serial adders discussed in Section 4.2.4. The weights of the network are all ternary values, however for the activations of the network, 4 fractional bits were used leaving 12 integer bits. Each layer is followed by batch normalization, containing vectors of constants, and also has a scaling factor which are all floating-point values. These constant values were combined in floating-point, as discussed in Section 4.2.5 and Equation 4.4, then quantized with 6 fractional bits leaving 10 integer bits. The number of fractional bits was obtained experimentally to achieve maximum precision without the risk of overflow. This was relatively simple to apply to the network and its impact on the performance achieves the same result as the floating-point scaling reported in Table 5.1. The design was then

Table 5.8: FPGA CNN Architecture blocks

Operation	Image Size In	Channel In	Channel Out
Buffer	32x32	3	3
Conv	32x32	3	64
Scale and Shift	32x32	64	64
Buffer	32x32	64	64
Conv	32x32	64	64
Scale and Shift	32x32	64	64
Buffer	32x32	64	64
Max Pool	32x32	64	64
Buffer	16x16	64	64
Conv	16x16	64	128
Scale and Shift	16x16	128	128
Buffer	16x16	128	128
Conv	16x16	128	128
Scale and Shift	16x16	128	128
Buffer	16x16	128	128
Max Pool	16x16	128	128
Buffer	8x8	128	128
Conv	8x8	128	256
Scale and Shift	8x8	256	256
Buffer	8x8	256	256
Conv	8x8	256	256
Scale and Shift	8x8	256	256
Buffer	8x8	256	256
Max Pool	8x8	256	256
FIFO	4x4	256	256
MuxLayer	4x4	256	4096
Dense	1x1	4096	128
Scale and Shift	1x1	128	128
MuxLayer	1x1	128	128
Dense	1x1	128	10

implemented using Chisel3 [4] which facilitated the generation of irregular adder trees. The CNN was partitioned into hardware blocks discussed in previous sections to stream the images through the network as shown in Table 5.8.

As discussed in Section 4.2.4, convolutional layers can be implemented with word or bit serial adders if the throughput is low enough. The max pool layer after the 2nd convolutional layer shrinks the image from 32×32 to 16×16, dropping the throughput by 75%. The 3rd and 4th convolutional layers therefore have 4 cycles to compute each

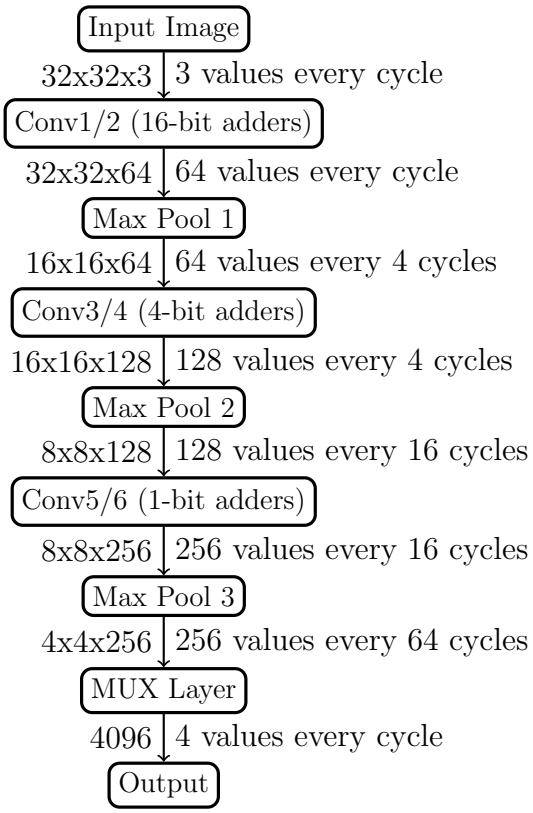


Figure 5.1: Impact of Max Pool Layers

addition. As 16 bit fixed point representation is used for the activations, the 3rd and 4th convolutional layers can use 4 bit word serial adders requiring a quarter of the area. Similarly for the fifth and sixth convolutional layers, the throughput required is only one output every sixteenth cycle. Hence, bit serial adders are used to compact the design to $\frac{1}{16}$ of the area of a full adder while maintaining sufficient throughput to ensure a fully pipelined design. This avoids the adders idling by creating hardware to compute the result over numerous cycles.

Figure 5.1 demonstrates how this impacts the design at a high level. The left side shows the image size, while the right side shows the throughput required. This structure allows the pruned adder tree to be implemented efficiently while still exploiting unstructured sparsity and common subexpressions in the design to reduce the area required. It is ineffective to implement the dense layer using CSE as discussed in Section 4.2.6.

Table 5.9: Ops needed to compute

Layer	Num Mults	Num Mults	With Sparsity	With CSE
Conv1	32*32*3*3*3*64	1769472	716800	630784
Conv2	32*32*3*3*64*64	37748736	8637440	4653056
Conv3	16*16*3*3*64*128	18874368	4559616	2654720
Conv4	16*16*3*3*128*128	37748736	9396480	5213440
Conv5	8*8*3*3*128*256	18874368	4656768	2504640
Conv6	8*8*3*3*256*256	37748736	9356736	4731840
Dense	4096*128	524228	524228	1048456[1]
SM	128*10	1280	1280	2560[1]
Total	153289924	153 MMACs/Image	38 MMACs/Image	21 MOps/Image

Table 5.9: [1] Obtained by converting one MACs to two Ops

5.1.3 The Aggregated Network

The VGG-7 network blocks for the FPGA design are described in Table 5.8. This architecture is designed to accept a single pixel each cycle. Given an image size of $W \times W$, W^2 cycles are therefore required to stream each image in. Hence, classifying a new image every W^2 cycles and assuming a clock frequency of f_{clk} gives a throughput of $\frac{f_{\text{clk}}}{W^2}$ classifications/second. The ternary MACs required to compute the VGG-7 style network described in Table 5.2 for a single image is given in Table 5.9. Operations in Max Pool and batch normalization are orders of magnitude lower than the convolution and not included in this total.

Hence for a fully floating-point implementation with the equivalent throughput to this FPGA implementation, this network requires a total of 153 MMAC/Image $\times$ $\frac{f_{\text{clk}}}{W^2}$ Images/sec $\times$ 2 Ops/MAC. In the case of a network with ternary weights, the multiplication is with -1,0 and 1. Table 5.9 then shows the MMACs after accounting for sparsity, reducing the cost per image to 38 MMACs. As discussed in Section 4.2.4, this can be implemented with only adds and subtracts with the CSE structure. The multiplication is performed in the dense layers but no longer in the convolutional layers and hence the final column in Table 5.9 shows the MOps needed on the FPGA for a single image, counting 1 MAC as 2 Ops. The actual computation performed on the FPGA with CSE is therefore 21 MOps/Image $\times$ $\frac{f_{\text{clk}}}{W^2}$ Images/sec. With these parameters and the implementation of the dense layers described in section 4.2.6, the largest dense

layer needs $4096 \times 128 \times 2b = 1Mb$ of storage for the weights as each weight is only 2 bits. As this is computed at a rate of 4 per cycle, the bandwidth for this memory is required to be $4 \times 128 \times 2b = 1Kb$ each cycle. Given that a BRAM on a Xilinx device typically has a output bandwidth of 64 bits each cycle, this implies just 16 BRAMs are needed to store the weights, each storing 64kb of data.

5.1.4 Hardware Results

The entire system is implemented on an AWS F1 instance and achieves a frequency of 125 MHz. It is a loopback application which passes the CIFAR10 images in from the C application using DPDK libraries [21], onto the FPGA via PCIe, though the network and back to the C program. The relatively low clock frequency of 125 MHz was necessary due to routing congestion that was observed when using tighter clock constraints. The bottleneck is due to routing between the convolutional layers which remained despite constraining the critical path onto the same Super Logic Region (SLR). Due to the windowing for the convolution in the buffer layer, a large amount of routing to different CLBs for the input of the matrix multiplication is required. Due to the limits in routing resources, it is difficult to fanout the wide bus to all adders that require that input. The critical path is in the layers that have high numbers of inputs that are going to a variety of locations. Conv2 has 9*64 inputs with 16 bit bus width. This is a wide bus width, but requires a smaller number of unique locations and hence the granularity of the routing can be coarse. Conv4 has 9*128 inputs with 4 bit bus width and Conv6 has 9*256 inputs with 1 bit bus width. With a larger convolutional layer, more fanout is required at these points. The critical path in this design is between the two modules in bold in Table 5.8.

The synthesisable Verilog register transfer level (RTL) code was generated in Chisel3, with custom modules being used to invoke components from the Vivado IP Core, such as the FIFOs. The design was developed using Vivado 2018.2. The hardware was verified with an input image passed into the design and obtaining identical output as

Table 5.10: Vivado size hierarchy

Block	LUTs/1182240	FFs/2364480
Conv1	3764 (0.3%)	10047 (0.4%)
Conv2	40608 (3.4%)	71827 (3%)
Conv3	55341 (4.7%)	56040 (2.4%)
Conv4	111675 (9.4%)	110021 (4.6%)
Conv5	73337 (6.2%)	79233 (3.4%)
Conv6	127932 (10.8%)	139433 (5.9%)
All Conv	535023 (45.2%)	631672 (26.7%)
Dense	12433 (1.1%)	19295 (0.8%)
SM	500 (0.04%)	442 (0.01%)
Whole CNN	549358 (46.5%)	659252 (27.8%)
Whole design	787545 (66.6%)	984443 (41.6%)

verified against a Python script used to compute the fixed point performance of the network in Table 5.1. The code for the implementation is publicly available on github[1].

Table 5.10 shows the resources used by various layers of the network in the implemented design. In total, 66.6% and 41.6% of the LUT and FF resources were utilized.

For $f_{\text{clk}} = 125$ MHz, the total theoretical Ops required for an equivalent dense floating-point implementation is 153 MMAC/Image $\times \frac{125MHz}{32^2}$ Images/sec $\times$ 2 Ops/MAC = 37.3 TOps/sec. For this implementation, multiplications in the convolutional layers are not required and a significant portion of the operations can be removed. Hence, in practice only approximately 2.5×10^{12} Adds/sec are necessary. The implemented system achieves the throughput of classifying 122k images/sec and a latency of 29 μs as only a fraction of the PCIe bandwidth is needed. This is including a DPDK [21] virtual ethernet interface to send and receive packets from the FPGA. Table 5.11 gives a comparison of this work with previously reported results for CIFAR10. This shows the properties of different implementations. There are TOps column in this table to show the actual ops computed on the FPGA, the logical ops and the equivalent floating-point ops for the same sized network. As these operations require very few bits, multiplying and accumulating with 1 or 2 bit numbers are typically integrated in a single logical statement. For this reason the actual logical operations are also shown. The values in

[1] github.com/da-steve101/aws-fpga.git and github.com/da-steve101/binary_connect_cifar.git

order for the TOps column are Actual/Logical/Equivalent (A/L/E). Table 5.11 shows that this method significantly improves both latency and throughput compared with the previous best reported results for FPGAs. The latency reported is for the entire system.

Remarkably, this is achieved with better accuracy compared with previous work as much higher precision activations are used. Despite the low frequency, the current design already meets the performance of all existing comparable implementations even after normalizing for the large FPGA used in this work.

It should be noted that for a fully floating-point network, 37.3 TOps/sec would be required to achieve the same number of classifications. Only a fraction of these need to be executed in practice due to this methods compile-time optimisations as shown in the right two columns of Table 5.9. This shows the reduction is mostly due to the implementation taking advantage of unstructured sparsity of 75%. This is further reduced by the application of CSE to the adder tree to reduce the remaining operations by removing the multiplications and merging computations. Not reflected in the TOps but critical for the design is the 4 bit word serial adders and the bit serial adders which reduce area by factors of approximately 4 for layers 3 and 4 and 16 for layers 5 and 6, making this method feasible for the AWS F1 platform.

5.1.5 Comparison with Previous Work

The method proposed by Prost-Boucle et al. [67] achieves the previously best reported low precision throughput on an FPGA for CIFAR10. This work implements a VGG-7 style network with ternary weights and activations. The approach is similar to Baskin et al. [6] and Li et al. [50] at a higher frequency of 250 MHz and using hand written Register Transfer Language (RTL) as opposed to C-based High-Level Synthesis (HLS) tools. Their work is also the most similar to this method both in architecture and network choice, but they used an FPGA approximately 4× smaller. Compared with their work, this design achieves significantly higher FPS (122 k FPS vs 27 k FPS).

Accounting for the much larger FPGA used in this work, this improvement in throughput disappears as the frequency is 2× higher than achieved with this design. The reason for the comparable throughput despite the advantage of exploiting sparsity is the use of 16-bit activations in this work in contrast to ternary activations in Prost-Boucle et al. [67] and binary activations in Fraser et al. [24]. This design choice results in higher accuracy 90.9% on CIFAR10 compared to Prost-Boucle et al. with 86.7% or Fraser et al. with 88.7%.

The custom ASIC accelerator by Jouppi et al. [36], was designed with datacenter requirements in mind. Their work uses higher precision weights, meaning higher accuracy can probably be achieved. Due to the much higher frequency of 700 MHz compared to 125 MHz and the difficulty of comparing the amount of hardware used it is hard to make a meaningful comparison on throughput. Given the reported die area of 331mm^2 and the Ultrascale of 2256mm^2, the ASIC is significantly smaller. Hence, in terms of throughput alone, the TPU is superior to this work. However, the latency achieved by this design is very low. While the TPU does not have a benchmark for CIFAR10, they quote a latency of around 10ms to achieve the peak throughput with a batch size of 250 images. This is nearly three orders of magnitude more than the implementation in this work. For latency sensitive applications, this is a significant difference. Additionally, the TPU is not commercially available for purchase and is accessible only through Google Cloud.

YodaNN [3] create a very small ASIC with an area of 1.9mm^2. They use binary weights and 12 bit activations and hence achieve a very high throughput for the area they used. This achieves lower precision than the TPU and hence would likely lose some accuracy. However, their throughput for the area is nearly 3× that of the TPU and, using 65 nm technology, more than double the TPUs 28nm.

Venkatesh et. al. [85] use 14nm technology to claim a peak speed of 2.5 TOps/sec with a size of 1.09mm^2. This uses ternary activations and a half precision floating-point, meaning it will likely have higher accuracy than YodaNN.

Table 5.11: Comparison of CNN inference implementations for CIFAR10 where reported for ASICs (top) and FPGAs (bottom).

Reference	Hardware (mm^2,nm,LE[3]/LC[3] $\times 10^6$)	Precision (wghts, actv)	Freq. [MHz]
[85]	ASIC(1.09,14,–)	(2,16[2])	500
[3]	ASIC(1.9,65,–)	(1,12)	480
[36]	ASIC(331,28,–)	(8,8)	700
[6]	5SGSD8(1600,28,0.7)	(1,2)	105
[50]	XC7VX690(1806.25,28,0.7)	(1[1], 1)	90
[51]	5SGSD8(1600,28,0.7)	(1,1)	150
[67]	VC709(1806.25,28,0.7)	(2,2)	250
[83]	ZC706(961,28,0.35)	(1,1)	200
[24]	KU115(1600,20,1.45)	(1,1)	125
[86]	KU115(1600,20,1.45)	(1, 1)	200
This work	VU9P(2256.25,20,2.6)	(2,16)	125

[1] First layer is fixed point, [2]floating-point, [3] LE and LC are from Xilinx or Altera documentation of the FPGAs

Wang et. al. [86] propose a method of training binary neural networks that is favourable for FPGAs. By designing a network that efficiently uses FPGA architecture, they are able to achieve a high density. However, in their approach only certain sections of the network is unrolled due to area constraints. It is unclear from their paper the throughput that this implementation achieves.

To summarise, the method proposed in this work achieves the highest throughput and the lowest latency reported on an FPGA platform so far, significantly reducing the gap between the highly customized ASIC architectures.

Figure 5.2 shows a comparison of accuracy and throughput normalized for Logic Elements (LE) and Logic Cells (LC). This figure compares the available FPGA implementations on the CIFAR10 dataset. Due to the higher precision activations chosen in this work, the accuracy achieved by this design is significantly higher. Despite the higher precision used, this design keeps up in throughput per LE or LC with other much lower precision implementations.

For figure 5.2, all the implementations do not exploit sparsity of CSE as this work does. These implementations typically rely on storing the weights in memory and loading

Table 5.12: Comparison of CNN inference implementations for CIFAR10 where reported for ASICs (top) and FPGAs (bottom).

Reference	Latency A/E[1]	TOps/sec	FPS	Accuracy
[85]	2.5/2.5	–	91.6%[2]	
[3]	–	1.5/1.5	434	–
[36]	≈10 ms	86/86[3]	–	–
[6]	–	–	1.2 k[2]	84.2%
[50]	–	7.7/7.7	6.2 k	87.8%
[51]	–	9.4/9.4	7.6 k[2]	86.31%
[67]	–	8.4/8.4	27 k	86.7%
[83]	283 μs	2.4/2.4	21.9 k	80.1%
[24]	671 μs	14.8/14.8	12 k	88.7%
[86]	–	–	–	85%
This work	29 μs	2.5/37.3	122k	90.9%

[1] Actual/Equivalent, [2]estimated, [3] 92 TOps/sec peak

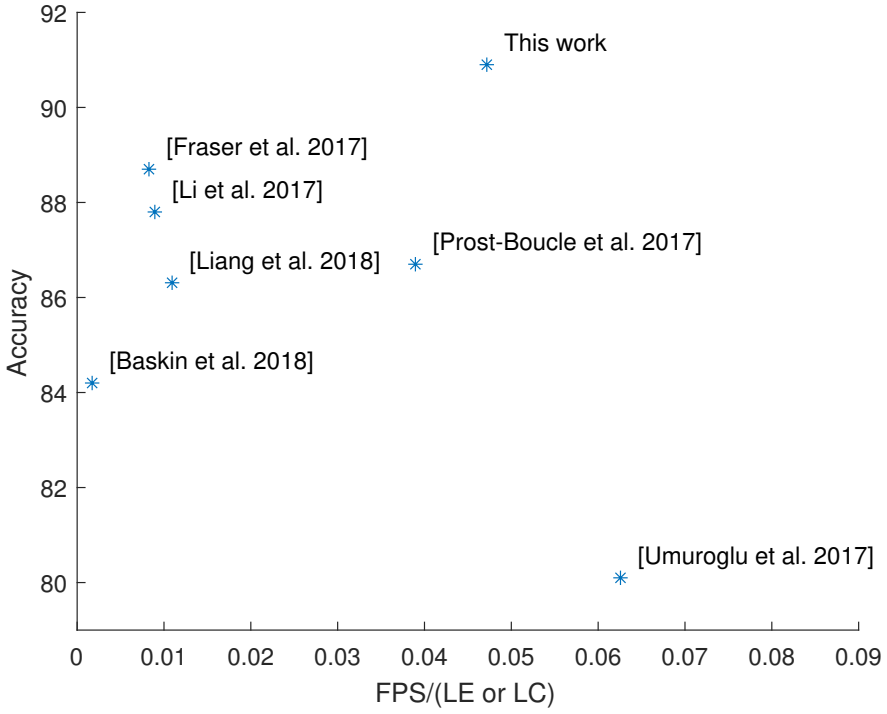


Figure 5.2: Comparison of FPGAs on CIFAR10

them when needed. Additionally they all use lower precision than this work. This is necessary to fit a reasonable sized network with high throughput on an FPGA. By

removing a large number of operations, this work is able to use higher precision and still fit a reasonably sized network.

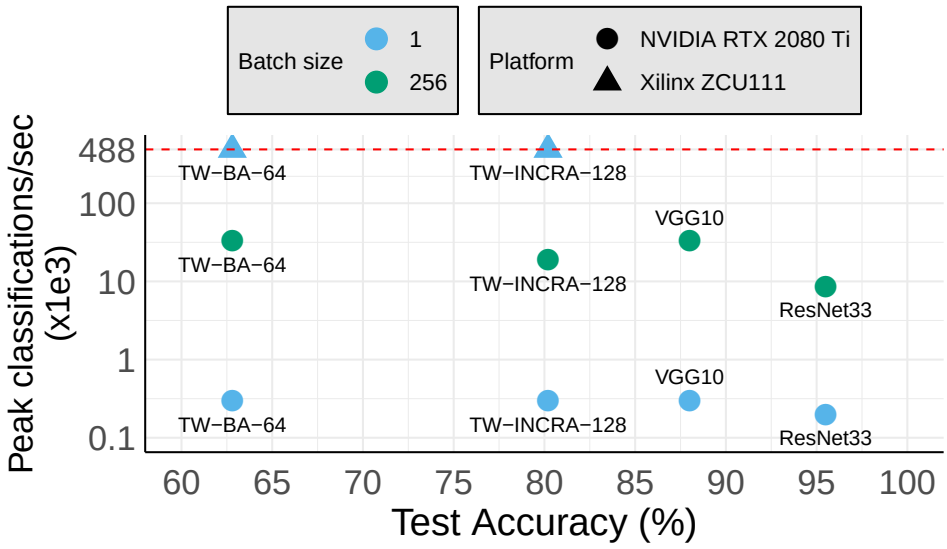


Figure 5.3: Peak classification throughput vs model accuracy on 24 modulation classes, measured on two modern hardware platforms. Note that increasing batch size also increases the response latency of the model.

5.2 RFSoC Case Study

In Automatic Modulation Classification (AMC), the goal is to accurately identify the modulation type used by a transmitting source based on RF samples taken from the environment [94]. AMC algorithms typically rely on mathematical techniques, where low-dimensional classifiers (*e.g.* support vector machines, K-nearest neighbours, linear models, etc) are trained on features that are carefully hand-crafted by domain experts [1, 10, 70]. Deep neural networks, on the other hand, allow these features to be learned during training, and have been shown to outperform the state-of-the-art, especially when there is a low signal-to-noise (SNR) ratio in the transmission channel [62, 93].

While research into deep learning methods for AMC has gained significant traction, there is far less research on realizing these machine-learning based systems as real-time hardware implementations. CNNs are typically compute-bound [35], and while domain-specific accelerators can offer markedly better performance over commodity general-purpose hardware, they are usually targeted towards computer vision applications where a classification throughput of tens to hundreds of classifications per second is sufficient. Figure 5.3 shows this effect on a modern NVIDIA RTX 2080 Ti GPU, which tops out

at a peak throughput of ≈ 30k classifications per second on the smaller models. In addition, a large batch-size plays a vital role in sustaining this throughput, which may not be feasible for a real-time implementation due to latency constraints. For Radio Frequency (RF) applications, the data-rate can be much higher on the order of hundreds of millions of samples per second, and hence, both latency and throughput of the AMC design are critical for sensing and responding to changes in the RF channel. FPGAs can play a vital role in realising these high-speed real-time systems, as they have been a popular choice in digital signal processing (DSP) applications for several decades now [81]. In addition, the programmable logic offered by FPGAs tie well into the world of machine learning [14, 25, 55], where models and algorithms are continuously evolving and pushing the research frontiers.

The Xilinx ZCU111 RFSoC [19] is a new FPGA-based radio platform targeted for research and development on Software-Defined Radio (SDR) applications. The platform is built on the Zynq Ultrascale+ SoC family, and offers runtime-programmable multi-gigasample RF transceivers and soft-decision forward-error-correction hardware blocks for building high-speed radio systems. In this section, the power of this new RFSoC platform is leveraged to build an end-to-end system for doing real-time automatic modulation classification using state-of-the-art machine learning methods. As seen from Figure 5.3, this method improves the classification throughput by several orders of magnitude on the RFSoC platform (batch size 1), while achieving competitive classification accuracy on a large number of modulation classes.

All network designs discussed in this section are trained and evaluated on an open-source dataset created by O'Shea et. al. [62]. The RadioML 2018.01A dataset[2] is a collection of raw I/Q samples that have been captured over-the-air using USRP devices as described in [62]. Each training sample is a time-series of 1024 I/Q sample pairs, accompanied by a label that identifies its modulation class. There are a total of 24 modulation classes recorded at 26 SNR levels, ranging from -20dB to +30dB in increments of 2dB. Each $\{modulation\ class, SNR\}$ pair has 4096 training examples, and hence, there are a total

[2] https://www.deepsig.io/datasets, *Accessed: July 2019*

of 2.56M labeled I/Q time-series examples in the entire dataset. The 24 modulation classes of the signals are: OOK, 4ASK, 8ASK, BPSK, QPSK, 8PSK, 16PSK, 32PSK, 16APSK, 32APSK, 64APSK, 128APSK, 16QAM, 32QAM, 64QAM, 128QAM, 256QAM, AM-SSB-WC, AM-SSB-SC, AM-DSB-WC, AM-DSB-SC, FM, GMSK and OQPSK. This dataset was divided into training (90%) and test sets (10%).

5.2.1 Related Work

Deep learning for RF applications is a relatively new field, and in particular, existing work on AMC has been restricted to model design and evaluation purely in software. Mendis et. al [59] compute the spectral correlation function (*i.e.* a cyclo-stationary feature extraction step) and implement a deep-belief network for classifying five modulation classes. In [54, 69], the authors report the effectiveness of various deep neural networks on ten modulation classes, and demonstrate several strategies that help reduce training time. Zhang et. al [91] propose a heterogeneous CNN / LSTM (long short-term memory) model that delivers high classification accuracy on eleven modulation classes. While inference runtime is reported for CNN networks (the order of a few seconds), the authors do not report runtime of the proposed heterogeneous models, which is likely to be longer due to the increased complexity in the models. O'Shea et al. [61, 62] design and evaluate two CNN models (VGG10 and ResNet33) and demonstrate competitive accuracy performance on 24 modulation classes. In all of these studies, runtime boundaries for real-time implementation are either not reported or are incomplete. This work aims to fill this research gap.

The models proposed in [62] – VGG10 and ResNet33 – were used as the baseline. Design strategies were explored to achieve a high-speed and resource-efficient FPGA implementation. In order to achieve this, experiments were performed with low precision variations of the VGG10 network and the tradeoff of computation with accuracy was explored. Due to the size of the ResNet33 model, implementing it on an FPGA with high throughput is difficult for two reasons: (1) the model size is too large to be spatially mapped to the FPGA fabric, which limits the achievable classification throughput

significantly, and (2) the residual connections create an unbalanced design which can result in bottlenecks if large amounts of on-chip memory are not available to store intermediate activations. Hence, the smaller and simpler VGG10 model was chosen instead for a real-time FPGA-based AMC implementation.

The VGG10 model has seven 1D convolutional layers followed by three dense layers. All of the convolutional layers have a kernel size of three and a stride of one. The convolutions are followed by maxpool, batch normalization [33] and the ReLU activation layers. In [62], the first two dense layers use alpha dropout [40] followed by the SELU activation function. This training method was used for networks with floating-point precision, but swapped alpha dropout layers with batch normalization layers when training low-precision networks for improved training stability.

5.2.2 Training Method

All models are trained on the RadioML 2018.01A dataset. The batch size was set to 128, the initial learning rate to 10^{-3}, and to train for 250k steps. The learning rate was set to smoothly decay exponentially at a rate of 0.5 every 100k steps. The dataset was partitioned into a 90% – 10% split to create the train and test sets respectively, such that each {SNR, modulation class} pair had 3686 train and 410 test examples. For training, samples captured at $\geq$ +6dB SNR were used which is typically the minimum signal strength observed in most wireless communication systems. This improves the convergence and ensures that the CNN is able to achieve the best classification accuracy for typical real-time use cases. In addition, the teacher-student [2] training methodology was used to further improve accuracy. In this case, for all VGG10 networks, including floating-point implementations, a trained ResNet33 model was used as the teacher.

Ternary weight networks were quantized using the method described in Section 5.1, which results in a network with ternary weights and floating-point activations. For networks with quantized activations, the floating-point activation values were first

clipped between 0 and 1, and then computed

$$x' = \frac{round(x(2^k - 1))}{2^k - 1}$$

where x is an activation, and k is the number of bits to use for the quantization. This activation quantization is done after the ReLU of each layer.

5.2.3 Model Design

Using model design techniques described in this chapter, low precision CNNs are implemented and evaluated. Several different model sizes are explored where the goal is to maximize classification accuracy as much as possible while fully utilizing the available FPGA resources. Table 5.13 details all the models and their properties explored in this section.

The first four networks in Table 5.13 are trained and tested with floating-point weights and activations. All the networks with the prefix `TW-` are trained with ternary weights (*i.e.* {-1, 0, 1}). For all networks, $\nu = 0.7$ was used for the first layer, $\nu = 1.2$ for the remaining convolutional layers, and $\nu = 0.7$ for the two dense layers with ternary weights. Networks with `-BA-` in the model name refers to binary activations, where all activations from the layers are binary including the first two dense layers. Finally, `-INCRA-` refers to a network that was trained with incrementally increasing precision of activations, starting with 1b activations after the first layer. The incremental precision network, `TW-INRCA-128`, was trained with quantized activations for the first four layers to the following number of bits: 1, 2, 4, and 8. The next two convolutional layers do not have quantized activations and the final convolutional layer was quantized back to binary acitvations along with the first two dense layers. All networks have floating-point batch normalization variables. Neither the weights and activations of the final dense layer, nor the input data in all networks is ever trained with quantization.

Table 5.13: Properties of various models designed and explored in this paper.

Model Name	**Architecture**	**Precision[1] (Wts/Acts)**	**# params**	**# MACs**	**Accr[2]**
ResNet33 [62]	{ResBlock}×6, (FC, 128)×2, (FC/Softmax, 24)	32b/32b (FP)	507k (2.03Mb)	111m	95.5
VGG10 [62]	{(Conv, K3, 64), (MaxPool, S2)}×7, (FC, 128)×2, (FC/Softmax, 24)	32b/32b (FP)	102k (407Kb)	12.8m	88.0
VGG10-64	{(Conv, K3, 64), (MaxPool, S2)}×7, (FC, 512)×2, (FC/Softmax, 24)	32b/32b (FP)	381k (1.5Mb)	13.3m	89.6
VGG10-128	{(Conv, K3, 128), (MaxPool, S2)}×7, (FC, 512)×2, (FC/Softmax, 24)	32b/32b (FP)	636k (2.5Mb)	51.1m	90.9
TW-64	{(Conv, K3, 64), (MaxPool, S2)}×7, (FC, 512)×2, (FC/Softmax, 24)	2b/32b (FP)	381k (95kb)	13.3m	78.8
TW-96	{(Conv, K3, 96), (MaxPool, S2)}×7, (FC, 512)×2, (FC/Softmax, 24)	2b/32b (FP)	490k (123kb)	29.1m	82.4
TW-128	{(Conv, K3, 128), (MaxPool, S2)}×7, (FC, 512)×2, (FC/Softmax, 24)	2b/32b (FP)	636k (159kb)	51.1m	82.1
TW-BA-64	{(Conv, K3, 64), (MaxPool, S2)}×7, (FC, 128)×2, (FC/Softmax, 24)	2b/1b	102k (25kb)	12.8m	62.8
TW-BA-64-512	{(Conv, K3, 64), (MaxPool, S2)}×7, (FC, 512)×2, (FC/Softmax, 24)	2b/1b	381k (95kb)	13.3m	67.7
TW-BA-128	{(Conv, K3, 128), (MaxPool, S2)}×7, (FC, 512)×2, (FC/Softmax, 24)	2b/1b	636k (159kb)	51.1m	75.9
TW-INCRA-128	{(Conv, K3, 128), (MaxPool, S2)}×7, (FC, 512)×2, (FC/Softmax, 24)	2b/(variable)	636k (159kb)	51.1m	80.6

[1]FP = 32b floating-point; [2]Best accuracy at +30dB SNR

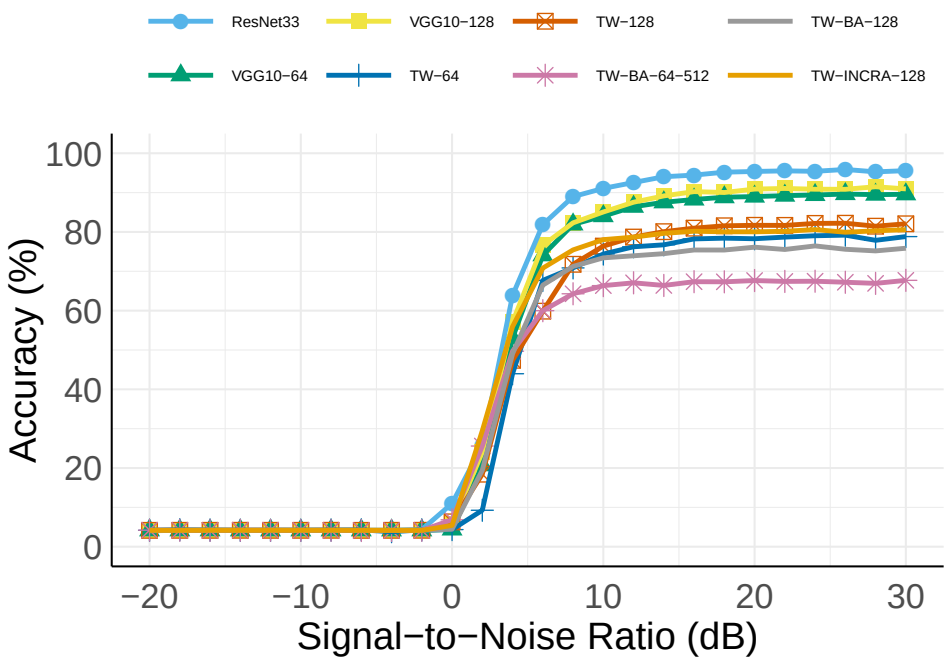
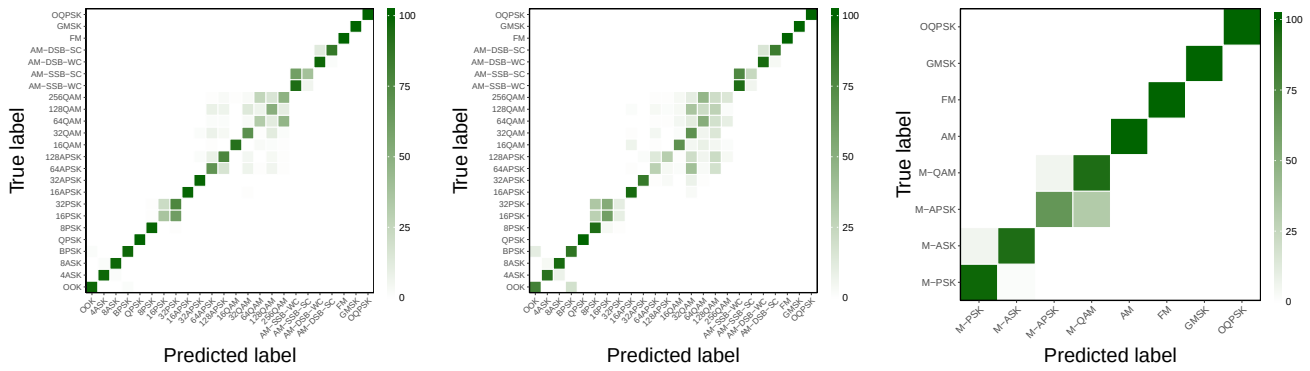


Figure 5.4: Accuracy (%) vs SNR (dB) with different models



(a) `ResNet33` (b) `TW-INCRA-128` (c) `TW-INCRA-128` with "super-classes"

Figure 5.5: Confusion matrix that demonstrates the effectiveness of signal modulation classification with ResNet33 and `TW-INCRA-128` models at +6dB SNR.

5.2.4 Model vs Accuracy Tradeoff

Higher complexity models offer improved performance in terms of accuracy but come at the cost of increased computation. Figure 5.4 shows the performance of different models at different SNR levels. As `ResNet33` has the largest number of parameters and

operations, it clearly outperforms the rest of the less expressive models. Unfortunately, as mentioned earlier, it is not feasible to implement a high-throughput `ResNet33` model on the target RFSoC ZCU111 platform. Similarly, the VGG10 models with floating-point precision are also infeasible due to the large area/performance overheads of floating-point hardware on FPGAs. Hence, the focus was on evaluating models that have ternary weights and fixed-point activations with varying number of filters.

The confusion matrices in Figure 5.5 show the distribution of incorrect classifications when tested with signals at +6dB SNR. The distribution illustrates that, intuitively, the difficultly in classification lies in similar higher-order modulation schemes. For example, a valid 256-QAM signal could appear identical to a 16-QAM signal given the right encoding, especially over varying SNR, which results in significant classication errors. This is apparent when comparing Figures 5.5a (`ResNet33`) and 5.5b (`TW-INCRA-128`), where most of the misclassifications in the hardware-friendly model are due to the high-order modulation schemes. If a radio application does not require fine-grained classification between modulation classes, much better overall classification performance can be achieved. In Figure 5.5c, "super classes" of different modulation schemes were created, which show that the classification accuracy across these "super classes" is almost 100% with a hardware-friendly model like `TW-INCRA-128`. The misclassification between M-ASK and M-PSK is due to OOK and BPSK modulation classes, which are very similar binary modulation schemes.

5.2.5 Quantization Error

The networks trained in Table 5.13 use 32b floating-point for the batch normalization variables and the final dense layer. During inference, the batch normalization variables can be multiplied into the scaling factors for the convolution to give an equation $bn(x) = ax + b$, where a and b are known constants that can be pre-loaded into the hardware design. Note that some networks (*e.g.* `TW-64`) also use floating-point activations. In order to avoid implementing floating-point blocks for portions of the computation, the calculations are done in fixed-point instead. It was empirically

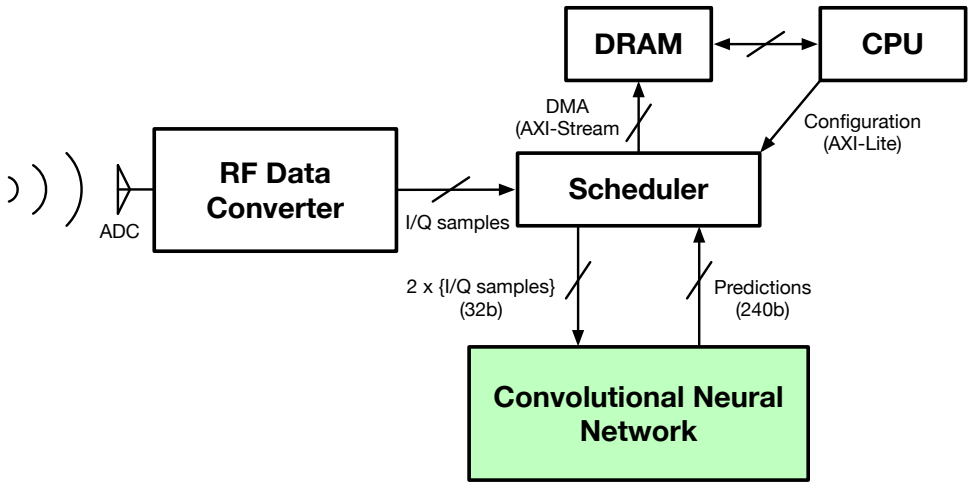


Figure 5.6: High-level view of the final system implementation on the ZCU111 RFSoC platform.

determined that these models deliver the best accuracy when 6 fractional bits in the activation layers, and 8 fractional bits for the batch normalization variables are used. Minimal numerical difference at the output was observed, typically around 2% for the class with the largest output activation value. Table 5.14 demonstrates that this proposed precision design strategy provides enough bits for stable implementations, such that any difference in accuracy is essentially training noise.

Table 5.14: Model accuracies on test set before/after quantization measured at +30dB SNR.

Model	**Accuracy (%)**	**Quantized Accuracy (%)**
TW-64	78.8	78.7
TW-96	82.4	81.1
TW-128	82.1	81.7
TW-BA-64	62.8	63.1
TW-BA-64-512	67.7	67.4
TW-BA-128	75.9	75.9
TW-INCRA-128	80.6	80.2

5.2.6 Scheduler

Figure 5.6 shows a high-level block diagram of the final implementation on the RFSoC ZCU111 platform, which is detailed in this and the following sections.

The scheduler orchestrates any data movement between the ADC, the CNN-based AMC core, and the Zynq Ultrascale host CPU. Data is communicated using the lightweight AXI4 stream protocol, where I/Q samples and prediction outcomes are sent as packets. The scheduler was designed as a configurable Xilinx IP core, allowing a user to easily reconfigure data-widths of signals if needed.

5.2.7 Auto-generator

In the course of this work, a Python3 package was developed, which has since been published as an open-source package in the pip3 packages repository called *twn_generator*. It is a Python implementation of the work previously implemented in Chisel3 for Section 5.1 with additional features that improve the hardware-design productivity significantly.

The Python package provides a general-purpose API that accepts matrix inputs expressed as comma-separated values (CSVs). Since convolutions can be represented as matrix-vector multiplications, the Python package can be easily generalized to evaluate any CNN model configuration, so long as the input matrix is conditioned appropriately. The package then auto-generates the desired Verilog implementations of the CNN layers, which can then be synthesized with the FPGA toolchains to create the final hardware implementation.

Before generating the Verilog, the package also performs Common Subexpression Elimination (CSE) on the input matrix in order to encourage result-sharing and to alleviate computational redundancy in the network. In experiments for this thesis, it was observed that this CSE step can reduce the computational requirements by as much as 55% of the original network, which is crucial for deriving a highly pipelined

architecture proposed in this work. The CSE outputs an irregular adder/subtractor tree that computes a single output pixel of the convolution. The CSE is independent of the activation precision or the throughput. The output tree from the CSE, along with the specified precision and method of adding the inputs, is then used to generate the RTL implementation of the computation.

Once the trained weights from a network are conditioned into a CSV file, generating the Verilog to compute the convolutions is less than 40 lines of Python code. The tree can also be fed into a C generator for quick verification of the test set results, which offered a substantial runtime boost to the evaluation phase. As of now, the package only supports the auto-generation of convolution layers, but an open-source GitHub repository [3] is provided that contains Verilog source of other neural network components (*e.g.*maxpooling). The maxpool and the dense layers of the VGG10 network are implemented in Verilog in a way similar to that described in Section 5.1. As the activations are quantized for these networks too, the implementation of batch normalization and ReLU is slightly different.

5.2.8 Implementation of Quantization, Batch Normalization and ReLU

During inference, the batch normalization layer computes $Y = ax + b$, which requires a fixed-point multiplication for each output channel of the convolutional layer and is frequently mapped to a DSP block on the FPGA. The batch normalization layer was merged with the ReLU function, where $relu(x) = x \times (x > 0)$ was computed. For quantized activations with k bits, the batch normalization and ReLU are implemented as:

[3] github.com/da-steve101/radio_modulation

$$Y_i = \begin{cases} 1, & \text{if } a_i x_i + b_i > 1 \\ \frac{round((a_i x_i + b_i)(2^k - 1))}{2^k - 1}, & \text{if } 0 \leq a_i x_i + b_i \leq 1 \\ 0, & \text{if } a_i x_i + b_i < 0 \end{cases} \tag{5.2}$$

On the FPGA, to compute the results with integers requires multiplying with $2^k - 1$ where necessary. The factor of $2^k - 1$ was incorporated into the batch normalization variables, a and b, prior to transforming them to fixed-point numbers.

5.2.9 Throughput Matching

As discussed in Section 4.3.6, the throughput of a pipelined CNN implementation is affected drastically by each maxpool layer. In a VGG10 model with 1D convolutions, the throughput drops by a factor of two after each maxpool layer, since each subsequent layer expects a smaller signal length at the input due to the maxpool operation. This drop in throughput would result in idle cycles for fully pipelined designs, where the hardware is simply waiting for new data to arrive before it is able to compute the next outputs. The number of idle cycles is exacerbated deeper down the network, and without throughput matching, the last convolution layer in VGG10 will be sitting idle for ≈97% of the time, which is a significant underutilization of the available hardware resources.

This property of the network can be exploited by doing careful throughput matching. Since the architecture of the CNN model is known from the beginning, a priori information is also available on the throughputs produced by each layer in the network. Additional precision can be allocated to the activations between layers in such a way that the number of idle cycles is minimized, or completely eliminated. For example, if a {convolution + maxpool} layer produces 2b activations, instead of using 2b adders in the subsequent layer, bit-serial adders can be used that compute each new output over 2 cycles. Precision can keep increasing in this fashion after each maxpool layer, which gives two positive outcomes: (1) the increased precision improves the accuracy of

the network, giving higher-quality classifications at the same throughput, and (2) the implementation can continue to use the bit-serial arithmetic units, which lowers the hardware utilization cost to the equivalent of binary activation networks. This technique essentially delivers an accuracy boost without any significant hardware overheads on the RFSoC platform, as observed in Table 5.15 and Table 5.14 when comparing `TW-BA-128` and `TW-INCRA-128` ($\approx$5% accuracy boost with $<$1% hardware overheads). Incrementally increasing precision in this way is a core contribution of the work submitted to FPT-19 [78].

Throughput matching in the dense layers was also done. However, instead of increasing precision, the datapath was unrolled in the dense layers such that its throughput matched the arriving data rate at its input. On the RFSoC platform, the size of the dense layers was increased from 128 in the baseline VGG10 model to 512 in these proposed models without suffering from over-utilization. When mapped naïvely, the last two dense layers in the baseline implementation sat idle for 75% of the time, which is reduced to zero in the proposed models. In addition, the final resource utilization is overall more balanced, as more DSPs and BRAMs are utilized to compute activations and store weights of dense layers respectively.

5.2.10 Functional Implementation on the ZCU111

While accuracies for the proposed models are trained and reported on the RadioML dataset for benchmarking purposes, a fully-functional implementation on the ZCU111 would require an end-to-end design where data captured from the on-chip ADCs is used for training. This is because subtle circuit-level discrepancies can destroy classification accuracy of models trained on a dataset that is not platform-specific.

In order to work towards this goal, a signal generator IP core that is capable of sending modulated signals over the ZCU111's DACs has also been created. On the receiver end, an I/Q sample collector IP block designed to help capture the data was implemented. Both IP cores are runtime configurable using the a lightweight AXI4-Lite interface (*e.g.*

modulation class to transmit, number of consecutive I/Q samples to collect, etc). As of now, the modulator IP core is capable of transmitting four different modulation classes: BPSK, QPSK, 8PSK, and QAM16. In the future, the aim is to extend this to cover other popular modulation classes, and also create a comprehensive dataset like RadioML, but platform-specific to the ZCU111 board. The goal is to motivate research and development of radio applications on the RFSoC platform by lowering the engineering cost associated with data collection on the board.

5.2.11 Methodology

Vivado 2018.3 was used to synthesize designs, the AXI4 communication framework was used to interface with the design through AXI4-Lite and AXI4-stream protocols and the PYNQ framework to manage, visualize, and verify functional correctness on Jupyter notebooks. For training, the Tensorflow framework was used to train and test various models, using the Ternary Weight Networks [48] approach for quantizing the weights.

In order to support the claim for real-time modulation classification, an I/Q sample rate of 500MHz was chosen and the CNN was designed to accept 2×I/Q samples each cycle with a 250MHz clock.

5.2.12 Resource Utilization

Table 5.15 shows the resource utilization for the various models compared in this work. The DSP usage is relatively constant as the convolutions do not use any DSPs. `TW-BA-64` shows significantly lower DSP usage as the dense layers have 128 outputs instead of 512 like in all the other models. For binary-activation networks, the DSPs are utilized by the batch normalization layer, which is fused at the output of the dense layers. For both dense layers with 512 outputs, this is 1024 multiplications, whereas for 128 outputs only 256 multiplications need to be performed. It should be noted that this is typically multiplying a 16 bit number with a 2 bit weight so mapping it to a DSP is optional for Vivado as this could be computed with CLBs. The largest network,

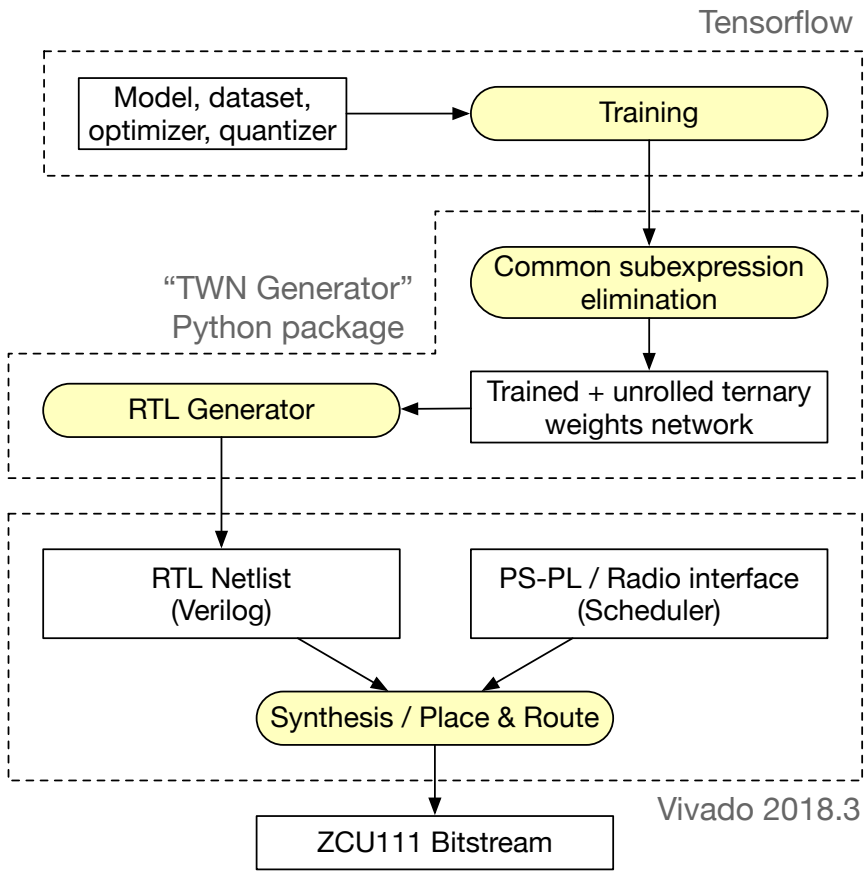


Figure 5.7: Completely automated toolflow for implementing a high-speed unrolled convolutional neural network for doing automatic modulation classification on the ZCU111 platform.

`TW-128`, fails to complete routing. All other designs met timing constraints with a 250 MHz clock.

The most interesting result in Table 5.15 is comparing `TW-BA-128` and `TW-INCRA-128`. The model `TW-BA-128` has binary activations throughout the network. This leads to lower hardware usage than `TW-128` but comes at the cost of a 5.8% accuracy drop to 75.9% as seen in Table 5.14. The model `TW-INCRA-128` differs from `TW-BA-128` as it has activations with increasing precision after each convolution layer (increasing from 1b to 16b, in powers of 2, and capped at 16b). This restores most of the accuracy, and is only 1.5% less accurate on the test set than `TW-128` (80.2%). However, looking at the hardware utilization, both `TW-BA-128` and `TW-INCRA-128` have essentially the same resource usage. This is because of the careful throughput matching carried out during

Table 5.15: FPGA implementation resource utilizations. All models were compiled out of context at 250MHz. Target device: `xczu28dr-ffvg1517-2-e`. [1] Failed to Route

Model	CLBs	LUTs	FFs	BRAMs	DSPs
TW-64	28k (53.5%)	124k (29.1%)	217k (25.5%)	524 (48.5%)	1496 (35%)
TW-96	47k (89.3%)	232k (54.7%)	369k (43.4%)	524 (48.5%)	1207 (28.3%)
TW-128[1]	51k (96.7%)	320k (75.3%)	506k (59.5%)	524 (48.5%)	1431 (33.5%)
TW-BA-64	11k (21.2%)	58k (13.7%)	92k (10.8%)	76 (7.0%)	704 (16.5%)
TW-BA-64-512	15k (27.6%)	80k (18.8%)	108k (12.7%)	521 (48.2%)	1471 (34.4%)
TW-BA-128	43k (80.7%)	234k (55.1%)	333k (39.2%)	523 (48.4%)	1408 (33.0%)
TW-INCRA-128	42k (80.2%)	211k (49.6%)	324k (38.1%)	512.2 (48.3%)	1407 (32.9%)

implementation, as described in Section 5.2.9. By designing the precision to increase with decreasing throughput requirements, the accuracy is restored by using hardware that would have otherwise been idle.

5.2.13 System Performance

To demonstrate the effectiveness of the method in practice, a simple dataset with four modulations on the RFSoC board was generated. A DAC and ADC on the RFSoC board were used in loopback fashion to generate BPSK, 8PSK, QAM16 and QPSK signals. A dataset was generated with 500k signals for each modulation type, each 1024 I/Q samples in length as recommended by O'Shea et. al. [62]. No noise was added to the signals. The dataset was then randomly partitioned into 90% train and 10% test resulting in 180K train examples and 20k test examples. As the dataset was small and simple, the baseline VGG10 network with ternary weights and fixed point activations was chosen. The network achieved zero test set error rate in less than an epoch. This

model was then put onto the board with the entire system as: Modulator → DAC → Coaxial Wiring → ADC → CNN → DMA → CPU.

This design accepted 2×I/Q samples from the ADC in each clock cycle and computed the prediction until 1024 I/Q samples had been provided before immediately starting the next one. A sample rate of 500MHz generated 488K signals of length 1024, each capturing 2.048 μs of data. This design was fully pipelined and matched this throughput. According to Xilinx documentation, the ADC has zero latency. With minor delays for some management logic, the entire network computed a classification in less than 2000 clock cycles from the first sample input or, with a 250MHz clock, 8μs. At a throughput of 488K classifications/second and a latency of 8μs, real-time modulation classification with high accuracy was achieved. The CPU has additional latency through the DMA, which is larger than the latency for the classification but still on the order of μs. With the 12.8 MMACs required to compute a single classification on the GPU, this network computed the equivalent of 6.28 TMACs on the FPGA.

Table 5.16 shows the utilization of different components of the system design. This shows there were significant portions of the FPGA unused in this design. Table 5.16 shows that at the top level, only 121k (30%) of the LUTs available were used by the entire design. The overhead of the rest of the modulator, RF-core and DMA engine was very small. Most of the resources were used by the CNN model implementation `TW-VGG`. Table 5.16 shows the resources required by each of the components in the CNN. `TW-VGG-Cx` show the resources needed to compute the convolutional adder trees generated by this Python package where `x` refers to the layer number. The table shows that layer 2 using 16 bit adders has the highest resource utilization. Layer 3 uses 8 bit adders, and only requires half the resources of layer 2 but has the same number of output channels. Subsequent layers use lower bitwidth adders, hence use fewer resources each time, until layer 6 and 7 which both use bit-serial adders. The first layer is the exception as it has far fewer inputs – only $2 \times 3 = 6$ – compared to the $64 \times 3 = 192$ for layer 2-7. `TW-VGG-Dx` show the usage needed by the dense layers. D1 and D2 use ternary weights and do not need DSPs to perform the multiplication, whereas D3 uses

16b fixed-point weights to compute, which is mapped using DSPs. The resources used by the batch normalization layers are all the same after each of the 7 convolutions and the first two dense layers. They compute a 16b fixed-point multiplication, and hence require 1 DSP block for each channel.

Table 5.16: FPGA implementation resource utilizations for RFSoC Design. Target device: `xczu28dr-ffvg1517-2-e`.

Element	LUTs	FFs	BRAMs	DSPs
Top	121k	198k	162	710
TW-VGG	109k	194k	152	708
TW-VGG-C1	3k	4k	0	0
TWVGG-C2	31k	45k	0	0
TW-VGG-C3	17k	26k	0	0
TW-VGG-C4	7k	14k	0	0
TW-VGG-C5	5k	8k	0	0
TW-VGG-C6	4k	5k	0	0
TW-VGG-C7	4k	5k	0	0
TW-VGG-D1	2k	4k	0	0
TW-VGG-D2	2k	4k	0	0
TW-VGG-D3	0k	0k	0	4
BN-CLYR ×7	0k	2k	0	64
BN-DLYR ×2	0k	4k	0	128
AXI-DMA	7k	5k	0	0
MODULATION-GEN	2k	2k	7	2
RF-CORE	3k	2k	0	0

5.3 Summary

In this chapter, two case studies were examined for the application of the architecture described in Chapter 4. Section 5.1 used the AWS F1 platform to create a high throughput and low latency image classifier for the CIFAR10 dataset. This section

demonstrated the effectiveness of CSE in reducing the computation required for the convolutional layers. Section 5.2 considered the AMC problem on the RFSoC ZCU111 development board. In addition to 16 bit activations, lower precision activations were explored and their effect on hardware area examined. Throughput matching was considered and hence an incremental precision scheme was proposed to exploit idle hardware in exchange for higher accuracy. Both sections demonstrated very high throughput and low latency implementations of TNNs.

Chapter 6

Conclusion

This thesis explored techniques to implement two machine learning methods on FPGAs. Hardware architectures were proposed to implement these methods with high throughput and low latency. The first method, NORMA, was discussed in Chapter 3. Chapter 4 proposed a methodology for implementing ternary neural networks on FPGAs. Chapter 5 applied the methodology to two different case studies demonstrating high throughput and low latency. The following two sections summarise the contributions presented in this thesis.

6.1 NORMA

In this thesis, a novel braiding technique for KAF sliding window algorithms was presented. Chapter 3 showed that when applied to a fixed-point implementation of NORMA, extremely low latency and high throughput can be achieved with similar learning ability to a floating-point implementation.

This was achieved through the application of a proposed technique called braiding. The formulation of NORMA was described in such a way so that the partial results could be computed and combined in a pipelined way. This mitigated the issue of the dependency in the NORMA update equations allowing high throughput to be achieved.

Compared to other approaches to implement KAFs, this method has significantly better throughput and latency for any other architecture on an FPGA.

6.2 Ternary Neural Networks

Chapter 4 described the unrolling of a convolution with ternary weights for very efficient inference due to its ability to exploit unstructured sparsity. Combined with the use of word or bit serial adders, this allows a very high throughput and low latency. While it is difficult to take advantage of sparsity on traditional computational platforms such as CPUs and GPUs, the method proposed has no overhead for exploiting sparsity as redundancies are removed at compile time.

Chapter 4 also demonstrated that the technique of Li et. al. [48] can directly tradeoff sparsity and accuracy by adjusting a single parameter. Exploiting this tradeoff reduces the area needed for the implementation due to increasing sparsity for a slight decrease in the accuracy.

As the proposed architecture does not require the image to be buffered, larger images such as in ImageNet could still be used. The main constraint in the implementation of this technique is the CNN size that will fit in a given hardware area. This method has the disadvantage that, compared with other approaches that can support larger networks, this method is restricted by the amount of hardware available. However, it very efficiently exploits unstructured sparsity and common subexpressions. This allows the design to remove a large portion of the computational task meaning, for a given hardware area, it achieves very high throughput. As the architecture is not dependent on buffering a batch of images to achieve high throughput a very low latency is achieved.

In Chapter 5, varying the precision of the activations to match the throughput was shown to dramatically improve accuracy while maintaining the same equivalent hardware usage as binary activations. This incremental precision approach to exploit the change in throughput after max pool layers improves efficiency usage of the hardware which is converted into higher accuracy.

6.3 Summary of Contributions

This thesis proposed the novel technique known as braiding for the implementation of NORMA onto an FPGA. Through the application of this technique, the highest throughput and lowest latency of a KAF on an FPGA was achieved.

The thesis also explored TNNs and how they might be implemented on a FPGA. By unrolling the weights, efficient exploitation of the sparsity allows the generation of compact high performance designs. It was also demonstrated that the application of CSE can further decrease the hardware required to implement convolutional operations. In the course of this work, a pip3 package twn_generator was developed to allow reproduction of results and the development of future techniques.

To demonstrate the effectiveness of the technique, two different datasets and platforms were used. The AWS-F1 platform is designed for datacenter applications and was used to classify images from the CIFAR-10 dataset. The RFSoC board which is designed for high performance radio applications was used to solve the automatic modulation classification problem.

6.4 Future Work

6.4.1 Kernel Methods

NORMA was chosen as an example application for braiding as it allows the flexibility to implement classification, regression and novelty detection algorithms. The braiding scheme described in Chapter 3 relies on properties of NORMA which are present in other algorithms. Generalisation to these problems would be promising candidates for further research.

The research in Chapter 3 used the same fixed point word length throughout each design. A more efficient scheme could vary the precision along the datapath to achieve similar accuracy with smaller word lengths. Further exploration of the precision required

for different components of the algorithm could lead to decreasing the hardware area needed to implement it.

One restriction that is notable from Table 3.1 is the restriction of the size of the dictionary. Implementations of NORMA with multiple FPGAs could enable larger dictionary sizes possibly improving the accuracy. This work has been made open source and the results reproducible, with a desire that others make improvements and utilise it in real-world machine learning problems.

6.4.2 Neural Networks

The approach of unrolling the entire inference computation has only recently become feasible due to the increased size of FPGAs, improved quality of tools and research into low precision networks. As FPGA capacity continues to increase, the unrolling of TNNs may become more favourable, particularly for small to medium neural networks. Future work could explore an implementation of ImageNet which utilises multiple FPGAs, in particular using Amazon f1.16xlarge instances which contain 8 FPGAs. Research into improving the merging of subexpressions for favourable routing in the convolutional layers should also be explored. Increasing the precision beyond ternary weights is possible if the CSE algorithm can cope with increased complexity.

One interesting avenue to explore would be developing hardware capable of exploiting this tree structure without needing to be reprogrammed. If hardware could be designed to directly execute the adder tree representing the convolution, a very high throughput and low latency could be achieved.

Finally, adding another layer of complexity with an architecture search guided by the hardware availability would mean a very complete package for automatic generation. Finding the largest model that fits on an FPGA with this technique would be very useful in the application of this technique.

As low precision training techniques improve and FPGA sizes increase, the unrolling methodology could become more effective for high throughput and low latency implementations as it allows larger networks to be used.

Bibliography

[1] Ameen Abdelmutalab, Khaled Assaleh, and Mohamed El-Tarhuni. "Automatic modulation classification based on high order cumulants and hierarchical polynomial classifiers". In: *Physical Communication* 21 (2016), pp. 10–18.

[2] Hande Alemdar, Vincent Leroy, Adrien Prost-Boucle, and Frédéric Pétrot. "Ternary neural networks for resource-efficient AI applications". In: *2017 International Joint Conference on Neural Networks (IJCNN)*. IEEE. 2017, pp. 2547–2554.

[3] Renzo Andri, Lukas Cavigelli, Davide Rossi, and Luca Benini. "YodaNN: An architecture for ultralow power binary-weight CNN acceleration". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 37.1 (2017), pp. 48–60.

[4] Jonathan Bachrach, Huy Vo, Brian Richards, Yunsup Lee, Andrew Waterman, Rimas Avižienis, John Wawrzynek, and Krste Asanović. "Chisel: constructing hardware in a scala embedded language". In: *DAC Design Automation Conference 2012*. IEEE. 2012, pp. 1212–1221.

[5] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. "Neural machine translation by jointly learning to align and translate". In: *International Conference on Learning Representations (ICLR)*. 2015.

[6] Chaim Baskin, Natan Liss, Evgenii Zheltonozhskii, Alex M Bronstein, and Avi Mendelson. "Streaming architecture for large-scale quantized neural networks on an FPGA-based dataflow platform". In: *2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE. 2018, pp. 162–169.

[7] Yoonho Boo and Wonyong Sung. "Structured sparse ternary weight coding of deep neural networks for efficient hardware implementations". In: *2017 IEEE International workshop on signal processing systems (SiPS)*. IEEE. 2017, pp. 1–6.

[8] Leo Breiman. "Random forests". In: *Machine learning* 45.1 (2001), pp. 5–32.

[9] P Cappello and Kenneth Steiglitz. "Some complexity issues in digital signal processing". In: *IEEE Transactions on Acoustics, Speech, and Signal Processing* 32.5 (1984), pp. 1037–1041.

[10] Claudomir Cardoso, Adalbery R Castro, and Aldebaro Klautau. "An efficient FPGA IP core for automatic modulation classification". In: *IEEE Embedded Systems Letters* 5.3 (2013), pp. 42–45.

[11] Philip K Chan and Salvatore J Stolfo. "Toward Scalable Learning with Non-Uniform Class and Cost Distributions: A Case Study in Credit Card Fraud Detection." In: *KDD*. Vol. 1998. 1998, pp. 164–168.

[12] Badong Chen, Songlin Zhao, Pingping Zhu, and Jose C Principe. "Quantized kernel least mean square algorithm". In: *IEEE Transactions on Neural Networks and Learning Systems* 23.1 (2011), pp. 22–32.

[13] Yu-Hsin Chen, Tushar Krishna, Joel S Emer, and Vivienne Sze. "Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks". In: *IEEE Journal of Solid-State Circuits* 52.1 (2017), pp. 127–138.

[14] Eric Chung, Jeremy Fowers, Kalin Ovtcharov, Michael Papamichael, Adrian Caulfield, Todd Massengill, Ming Liu, Daniel Lo, Shlomi Alkalay, Michael Haselman, et al. "Serving DNNs in real time at datacenter scale with project brainwave". In: *IEEE Micro* 38.2 (2018), pp. 8–20.

[15] Matthieu Courbariaux, Itay Hubara, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. "Binarized neural networks: Training deep neural networks with weights and activations constrained to +1 or -1". In: *2016 Conference on Neural Information Processing Systems (NIPS)*. 2016.

[16] Robert Daniels and Robert W Heath Jr. "Online adaptive modulation and coding with support vector machines". In: *Wireless Conference (EW), 2010 European*. IEEE. 2010, pp. 718–724.

[17] Yaakov Engel, Shie Mannor, and Ron Meir. "The Kernel Recursive Least Squares Algorithm". In: *IEEE Transactions on Signal Processing* 52 (2003), pp. 2275–2285.

[18] Julian Faraone, Nicholas Fraser, Giulio Gambardella, Michaela Blott, and Philip HW Leong. "Compressing Low Precision Deep Neural Networks Using Sparsity-Induced Regularization in Ternary Networks". In: *International Conference on Neural Information Processing Systems (NIPS)*. Springer. 2017, pp. 393–404.

[19] Brendan Farley, John McGrath, and Christophe Erdmann. "An all-programmable 16-nm RFSoC for Digital-RF communications". In: *IEEE Micro* 38.2 (2018), pp. 61–71.

[20] Cèsar Ferri, José Hernández-Orallo, and Peter A Flach. "A coherent interpretation of AUC as a measure of aggregated classification performance". In: *Proceedings of the 28th International Conference on Machine Learning (ICML-11)*. 2011, pp. 657–664.

[21] Linux Foundation. *Data Plane Development Kit (DPDK)*. 2015. URL: http://www.dpdk.org.

[22] Nicholas J Fraser and Philip H.W. Leong. "Kernel Normalised Least Mean Squares with Delayed Model Adaptation". In: *ACM Transactions on Reconfigurable Technology and Systems (TRETS)* (2020).

[23] Nicholas J Fraser, Duncan JM Moss, JunKyu Lee, Stephen Tridgell, Craig T Jin, and Philip HW Leong. "A Fully Pipelined Kernel Normalised Least Mean Squares Processor For Accelerated Parameter Optimisation". In: *Proc. International Conference on Field Programmable Logic and Applications (FPL)*. 2015.

[24] Nicholas J Fraser, Yaman Umuroglu, Giulio Gambardella, Michaela Blott, Philip Leong, Magnus Jahre, and Kees Vissers. "Scaling binarized neural networks on reconfigurable logic". In: *Proc. 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*. ACM. 2017, pp. 25–30.

[25] Brian Gaide, Dinesh Gaitonde, Chirag Ravishankar, and Trevor Bauer. "Xilinx Adaptive Compute Acceleration Platform: Versal TM Architecture". In: *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. ACM. 2019, pp. 84–93.

[26] Ben Graham. "Fractional max-pooling (2014)". In: *arXiv preprint arXiv:1412.6071* (2014).

[27] Albert Greenberg, James Hamilton, David A Maltz, and Parveen Patel. "The cost of a cloud: research problems in data center networks". In: *ACM SIGCOMM computer communication review* 39.1 (2008), pp. 68–73.

[28] Barbara Hammer and Kai Gersmann. "A note on the universal approximation capability of support vector machines". In: *Neural Processing Letters* 17.1 (2003), pp. 43–53.

[29] Song Han, Yu Wang, Shuang Liang, Song Yao, Hong Luo, Yi Shan, and Jinzhang Peng. "Reconfigurable Processor for Deep Learning in Autonomous Vehicles". In: (2017).

[30] David J Hand and Christoforos Anagnostopoulos. "A better Beta for the H measure of classification performance". In: *Pattern Recognition Letters* 40 (2014), pp. 41–46.

[31] Chun Hok Ho, Chi Wai Yu, Philip Leong, Wayne Luk, and Steven JE Wilton. "Floating-point FPGA: architecture and modeling". In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 17.12 (2009), pp. 1709–1718.

[32] Shen-Fu Hsiao, Ming-Chih Chen, and Chia-Shin Tu. "Memory-free low-cost designs of advanced encryption standard using common subexpression elimination for subfunctions in transformations". In: *IEEE Transactions on Circuits and Systems I: Regular Papers* 53.3 (2006), pp. 615–626.

[33] Sergey Ioffe and Christian Szegedy. "Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift". In: *International Conference on Machine Learning*. 2015, pp. 448–456.

[34] Yuanzhen Ji, Thomas Heinze, and Zbigniew Jerzak. "HUGO: real-time analysis of component interactions in high-tech manufacturing equipment (industry article)".

In: *Proceedings of the 7th ACM international conference on Distributed event-based systems*. ACM. 2013, pp. 87–96.

[35] Norman P Jouppi, Cliff Young, Nishant Patil, and David Patterson. "A domain-specific architecture for deep neural networks". In: *Communications of the ACM* 61.9 (2018), pp. 50–59.

[36] Norman P Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, et al. "In-datacenter performance analysis of a tensor processing unit". In: *Proc. 44th Annual International Symposium on Computer Architecture*. ACM. 2017, pp. 1–12.

[37] Alexandros Karatzoglou, Alex Smola, Kurt Hornik, and Achim Zeileis. "kernlab – An S4 Package for Kernel Methods in R". In: *Journal of Statistical Software* 11.9 (2004), pp. 1–20. URL: http://www.jstatsoft.org/v11/i09/.

[38] Jin Hee Kim, Brett Grady, Ruolong Lian, John Brothers, and Jason H Anderson. "FPGA-based CNN inference accelerator synthesized from multi-threaded C software". In: *2017 30th IEEE International System-on-Chip Conference (SOCC)*. IEEE. 2017, pp. 268–273.

[39] Jyrki Kivinen, Alexander J Smola, and Robert C Williamson. "Online learning with kernels". In: *Signal Processing, IEEE Transactions on* 52.8 (2004), pp. 2165–2176.

[40] Günter Klambauer, Thomas Unterthiner, Andreas Mayr, and Sepp Hochreiter. "Self-normalizing neural networks". In: *Advances in neural information processing systems*. 2017, pp. 971–980.

[41] Ron Kohavi. "A study of cross-validation and bootstrap for accuracy estimation and model selection". In: *Proceedings of the 14th international joint conference on Artificial intelligence-Volume 2*. Morgan Kaufmann Publishers Inc. 1995, pp. 1137–1143.

[42] Peter Kontschieder, Madalina Fiterau, Antonio Criminisi, and Samuel Rota Bulò. "Deep neural decision forests". In: *Proceedings of the Twenty-Fifth International Joint Conference on Artificial Intelligence*. AAAI Press. 2016, pp. 4190–4194.

[43] Max Kuhn. “Building Predictive Models in R Using the caret Package”. In: *Journal of Statistical Software, Articles* 28.5 (2008), pp. 1–26.

[44] Martin Kumm, Martin Hardieck, and Peter Zipf. “Optimization of Constant Matrix Multiplication with Low Power and High Throughput”. In: *IEEE Transactions on Computers* 66.12 (2017), pp. 2072–2080.

[45] Martin Kumm, Peter Zipf, Mathias Faust, and Chip-Hong Chang. “Pipelined adder graph optimization for high speed multiple constant multiplication”. In: *Circuits and Systems (ISCAS), 2012 IEEE International Symposium on*. IEEE. 2012, pp. 49–52.

[46] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. “Deep learning”. In: *Nature* 521.7553 (2015), pp. 436–444.

[47] Friedrich Leisch and Evgenia Dimitriadou. *mlbench: Machine Learning Benchmark Problems*. R package version 2.1-1. 2010.

[48] Fengfu Li, Bo Zhang, and Bin Liu. “Ternary weight networks”. In: *arXiv preprint arXiv:1605.04711* (2016).

[49] Guoqi Li, Changyun Wen, Zheng Guo Li, Aimin Zhang, Feng Yang, and Kezhi Mao. “Model-based online learning with kernels”. In: *IEEE transactions on neural networks and learning systems* 24.3 (2013), pp. 356–369.

[50] Yixing Li, Zichuan Liu, Kai Xu, Hao Yu, and Fengbo Ren. “A 7.663-TOPS 8.2-W energy-efficient FPGA accelerator for binary convolutional neural networks”. In: *FPGA*. 2017, pp. 290–291.

[51] Shuang Liang, Shouyi Yin, Leibo Liu, Wayne Luk, and Shaojun Wei. “FP-BNN: Binarized neural network on FPGA”. In: *Neurocomputing* 275 (2018), pp. 1072–1086.

[52] Weifeng Liu, Puskal P Pokharel, and Jose C Principe. “The kernel least-mean-square algorithm”. In: *IEEE Transactions on Signal Processing* 56.2 (2008), pp. 543–554.

[53] Weifeng Liu, Jose C Principe, and Simon Haykin. “Kernel Adaptive Filtering: A Comprehensive Introduction”. In: (2010).

[54] Xiaoyu Liu, Diyu Yang, and Aly El Gamal. "Deep neural network architectures for modulation classification". In: *2017 51st Asilomar Conference on Signals, Systems, and Computers*. IEEE. 2017, pp. 915–919.

[55] Nikhil Marriwala, Om Prakash Sahu, and Anil Vohra. "FPGA-Based Software-Defined Radio and Its Real-Time Implementation Using NI-USRP". In: *Field-Programmable Gate Array*. IntechOpen, 2017.

[56] Katsushige Matsubara, Kiyoshi Nishikawa, and Hitoshi Kiya. "A new pipelined architecture of the LMS algorithm without degradation of convergence characteristics". In: *Acoustics, Speech, and Signal Processing, 1997. ICASSP-97., 1997 IEEE International Conference on*. Vol. 5. IEEE. 1997, pp. 4125–4128.

[57] Naveen Mellempudi, Abhisek Kundu, Dheevatsa Mudigere, Dipankar Das, Bharat Kaul, and Pradeep Dubey. "Ternary Neural Networks with Fine-Grained Quantization". In: *arXiv preprint arXiv:1705.01462* (2017).

[58] Paolo Meloni, Alessandro Capotondi, Gianfranco Deriu, Michele Brian, Francesco Conti, Davide Rossi, Luigi Raffo, and Luca Benini. "NEURA ghe: Exploiting CPU-FPGA Synergies for Efficient and Flexible CNN Inference Acceleration on Zynq SoCs". In: *ACM Transactions on Reconfigurable Technology and Systems (TRETS)* 11.3 (2018), p. 18.

[59] Gihan J Mendis, Jin Wei, and Arjuna Madanayake. "Deep learning-based automated modulation classification for cognitive radio". In: *2016 IEEE International Conference on Communication Systems (ICCS)*. IEEE. 2016, pp. 1–6.

[60] Duncan JM Moss, Eriko Nurvitadhi, Jaewoong Sim, Asit Mishra, Debbie Marr, Suchit Subhaschandra, and Philip HW Leong. "High performance binary neural networks on the Xeon+ FPGA^{TM} platform". In: *Field Programmable Logic and Applications (FPL), 2017 27th International Conference on*. IEEE. 2017, pp. 1–4.

[61] Timothy J O'Shea, Johnathan Corgan, and T Charles Clancy. "Convolutional radio modulation recognition networks". In: *International conference on engineering applications of neural networks*. Springer. 2016, pp. 213–226.

[62] Timothy James O'Shea, Tamoghna Roy, and T Charles Clancy. "Over-the-air deep learning based radio signal classification". In: *IEEE Journal of Selected Topics in Signal Processing* 12.1 (2018), pp. 168–179.

[63] Prokopios Panagiotou, Achilleas Anastasopoulos, and A Polydoros. "Likelihood ratio tests for modulation classification". In: *MILCOM 2000 Proceedings. 21st Century Military Communications. Architectures and Technologies for Information Superiority (Cat. No. 00CH37155)*. Vol. 2. IEEE. 2000, pp. 670–674.

[64] Yeyong Pang, Shaojun Wang, Yu Peng, Nicholas J. Fraser, and P.H.W. Leong. "A Low Latency Kernel Recursive Least Squares Processor using FPGA Technology". In: *FPT*. 2013, pp. 144–151.

[65] F. Pedregosa et al. "Scikit-learn: Machine Learning in Python". In: *Journal of Machine Learning Research* 12 (2011), pp. 2825–2830.

[66] Fernando Perez-Cruz, Juan Jose Murillo-Fuentes, and Sebastian Caro. "Nonlinear channel equalization with Gaussian processes for regression". In: *Signal Processing, IEEE Transactions on* 56.10 (2008), pp. 5283–5286.

[67] Adrien Prost-Boucle, Alban Bourge, Frédéric Pétrot, Hande Alemdar, Nicholas Caldwell, and Vincent Leroy. "Scalable high-performance architecture for convolutional ternary neural networks on FPGA". In: *Field Programmable Logic and Applications (FPL), 2017 27th International Conference on*. IEEE. 2017, pp. 1–7.

[68] Jiantao Qiu, Jie Wang, Song Yao, Kaiyuan Guo, Boxun Li, Erjin Zhou, Jincheng Yu, Tianqi Tang, Ningyi Xu, Sen Song, et al. "Going deeper with embedded fpga platform for convolutional neural network". In: *Proc. 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. ACM. 2016, pp. 26–35.

[69] Sharan Ramjee, Shengtai Ju, Diyu Yang, Xiaoyu Liu, Aly El Gamal, and Yonina C Eldar. "Fast deep learning for automatic modulation classification". In: *arXiv preprint arXiv:1901.05850* (2019).

[70] Barathram Ramkumar. "Automatic modulation classification for cognitive radios using cyclic feature detection". In: *IEEE Circuits and Systems Magazine* 9.2 (2009), pp. 27–45.

[71] Mohammad Rastegari, Vicente Ordonez, Joseph Redmon, and Ali Farhadi. "Xnor-net: Imagenet classification using binary convolutional neural networks". In: *European Conference on Computer Vision*. Springer. 2016, pp. 525–542.

[72] Xiaowei Ren, Pengju Ren, Badong Chen, Tai Min, and Nanning Zheng. "Hardware Implementation of KLMS Algorithm using FPGA". In: *Neural Networks (IJCNN), 2014 International Joint Conference on*. IEEE. 2014, pp. 2276–2281.

[73] Shamnaz Riyaz, Kunal Sankhe, Stratis Ioannidis, and Kaushik Chowdhury. "Deep learning convolutional neural networks for radio identification". In: *IEEE Communications Magazine* 56.9 (2018), pp. 146–152.

[74] Bernhard Scholkopf and Alexander J. Smola. *Learning with Kernels: Support Vector Machines, Regularization, Optimization, and Beyond*. Cambridge, MA, USA: MIT Press, 2001. ISBN: 0262194759.

[75] Shai Shalev-Shwartz, Yoram Singer, Nathan Srebro, and Andrew Cotter. "Pegasos: Primal estimated sub-gradient solver for svm". In: *Mathematical programming* 127.1 (2011), pp. 3–30.

[76] David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, et al. "Mastering the game of Go without human knowledge". In: *Nature* 550.7676 (2017), p. 354.

[77] Emma Strubell, Ananya Ganesh, and Andrew McCallum. "Energy and Policy Considerations for Deep Learning in NLP". In: *arXiv preprint arXiv:1906.02243* (2019).

[78] Stephen Tridgell, David Boland, Philip HW Leong, and Siddartha. "Real-time Automatic Modulation Classification". In: *2019 International Conference on Field Programmable Technology (FPT)*. IEEE.

[79] Stephen Tridgell, Martin Kumm, Martin Hardieck, David Boland, Duncan Moss, Peter Zipf, and Philip HW Leong. "Unrolling Ternary Neural Networks". In: *ACM Transactions on Reconfigurable Technology and Systems (TRETS)* 12.4 (2019), p. 22.

[80] Stephen Tridgell, Duncan JM Moss, Nicholas J Fraser, and Philip HW Leong. "Braiding: A scheme for resolving hazards in kernel adaptive filters". In: *2015 International Conference on Field Programmable Technology (FPT)*. IEEE. 2015, pp. 136–143.

[81] Stephen M Steve Trimberger. "Three Ages of FPGAs: A Retrospective on the First Thirty Years of FPGA Technology: This Paper Reflects on How Moore's Law Has Driven the Design of FPGAs Through Three Epochs: the Age of Invention, the Age of Expansion, and the Age of Accumulation". In: *IEEE Solid-State Circuits Magazine* 10.2 (2018), pp. 16–29.

[82] Pınar Tüfekci. "Prediction of full load electrical power output of a base load operated combined cycle power plant using machine learning methods". In: *International Journal of Electrical Power & Energy Systems* 60 (2014), pp. 126–140.

[83] Yaman Umuroglu, Nicholas J Fraser, Giulio Gambardella, Michaela Blott, Philip Leong, Magnus Jahre, and Kees Vissers. "FINN: A framework for fast, scalable binarized neural network inference". In: *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. ACM. 2017, pp. 65–74.

[84] Steven Van Vaerenbergh and Ignacio Santamarıa. "A comparative study of kernel adaptive filtering algorithms". In: *Digital Signal Processing and Signal Processing Education Meeting (DSP/SPE), 2013 IEEE*. IEEE. 2013, pp. 181–186.

[85] Ganesh Venkatesh, Eriko Nurvitadhi, and Debbie Marr. "Accelerating deep convolutional networks using low-precision and sparsity". In: *Acoustics, Speech and Signal Processing (ICASSP), 2017 IEEE International Conference on*. IEEE. 2017, pp. 2861–2865.

[86] Erwei Wang, James J Davis, Peter YK Cheung, and George A Constantinides. "LUTNet: Rethinking inference in FPGA soft logic". In: *2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE. 2019, pp. 26–34.

[87] Naiyan Wang and Dit-Yan Yeung. "Learning a deep compact image representation for visual tracking". In: *Advances in neural information processing systems*. 2013, pp. 809–817.

[88] Maciej Wielgosz, Ernest Jamro, and Kazimierz Wiatr. "Highly efficient structure of 64-bit exponential function implemented in FPGAs". In: *Reconfigurable Computing: Architectures, Tools and Applications*. Springer, 2008, pp. 274–279.

[89] Ning Wu, Xiaoqiang Zhang, Yunfei Ye, and Lidong Lan. "Improving Common Subexpression Elimination Algorithm with A New Gate-Level Delay Computing Method". In: *Proceedings of the World Congress on Engineering and Computer Science*. Vol. 2. 2013.

[90] Chi Zhang and Viktor Prasanna. "Frequency domain acceleration of convolutional neural networks on CPU-FPGA shared memory system". In: *Proc. 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. ACM. 2017, pp. 35–44.

[91] Duona Zhang, Wenrui Ding, Baochang Zhang, Chunyu Xie, Hongguang Li, Chunhui Liu, and Jungong Han. "Automatic modulation classification based on deep learning for unmanned aerial vehicles". In: *Sensors* 18.3 (2018), p. 924.

[92] Shuchang Zhou, Yuxin Wu, Zekun Ni, Xinyu Zhou, He Wen, and Yuheng Zou. "DoReFa-Net: Training low bitwidth convolutional neural networks with low bitwidth gradients". In: *arXiv preprint arXiv:1606.06160* (2016).

[93] Siyang Zhou, Zhendong Yin, Zhilu Wu, Yunfei Chen, Nan Zhao, and Zhutian Yang. "A robust modulation classification method using convolutional neural networks". In: *EURASIP Journal on Advances in Signal Processing* 2019.1 (2019), p. 21.

[94] Zhechen Zhu and Asoke K Nandi. "Automatic Modulation Classification: Principles, Algorithms and Applications". In: (2015).

A

1 NORMA

All code for the reproduction of results for NORMA is available in the github repo at https://github.com/da-steve101/chisel-pipelined-olk.

2 TNNs

The github repo at https://github.com/da-steve101/binary_connect_cifar is written in chisel3 that generates the VGG network in Verilog and python scripts are used to verify the results and perform the CSE. This generated code is used in https://github.com/da-steve101/aws-fpga and has an interface modified from the AWS cl_sde example in branch sde_if_test.

Subsequent to this work a python3 package was developed, **twn_generator**. This separates out the CSE and generation from chisel to make it more accessible for developers.

3 Numerical example for Figure 3.5

To clarify figure 3.5, a numerical example is provided. Assume that $\nu_i = i$ or $\nu_1 = 1$, $\nu_2 = 2$ etc. These values are multiplied with the weights α. For simplicity, it will be assumed that $\alpha_i = 1$. The first 5 values are examples that are in beginning section of the existing dictionary. They are young enough so that they cannot be evicted. These results are just computed so $sum_1 = 3$ and $sum_2 = 7$. The 6th entry can only be

evicted if every new example is added. If the very first one is added, it is still possible for it to be evicted, otherwise it is not. We shall follow the case of where the first example is added and hence it is still possible to be evicted. In this case $sum_3 = 5$, $w_{6or7} = 6$, $w_{7or8} = 7$ and $w_{8or9} = 8$. On the next cycle, assume the example is not added to the dictionary. Therefore $Sum_R = 3 + 7 + 5 + 6 = 21$, $W_{D-1} = 7$ and $W_D = 8$.