Arbitrary Precision and Low Complexity Micro-Architectural Arithmetic Optimisations of Machine Learning Algorithms for Compute Bound and High-Performance Systems

by

James Philip Garland

Submitted for the degree of

Doctor of Philosophy
(Computer Science)

School of Computer Science and Statistics

THE UNIVERSITY OF DUBLIN, TRINITY COLLEGE DUBLIN

2021

Trinity College Dublin
Coláiste na Tríonóide, Baile Átha Cliath
The University of Dublin

Declaration

I declare that this thesis has not been submitted as an exercise for a degree at this or any other university and it is entirely my own work.

.....

James Philip Garland

Dated: 2021

Permission to Lend and/or Copy

I agree to deposit this thesis in the University's open access institutional repository or allow the Library to do so on my behalf, subject to Irish Copyright Legislation and Trinity College Library conditions of use and acknowledgement.

I consent to the examiner retaining a copy of the thesis beyond the examining period, should they so wish (EU GDPR May 2018).

.....

James Philip Garland

Dated: 2021

To my wife, Michelle,
my son, Toby and
my daughter, Zara,
without whom this thesis would have
probably been completed a year earlier ☺
I love you!

By the year 2600, the world's population would be standing shoulder to shoulder, and the electricity consumption would make the Earth glow red-hot.

Stephen Hawking, Tencent WE Summit, 2017

Abstract

ARTIFICIAL INTELLIGENCE is becoming ubiquitous and pervasive in our daily lives. *Machine learning (ML)*, a subset of *Artificial intelligence (AI)*, supplies more accurate internet searches, voice recognition in home appliances, tagging people in photos, object detection in videos, and driver assistance systems in vehicles. *Convolutional neural networks (CNNs)*, a subset of ML, process these images, videos and sometimes audio data. Captured and preprocessed by embedded *internet of things (IoT)* devices, CNN data are often processed in internet data centres or on local PCs with high-performance processors and acceleration cards, due to CNNs enormous energy, bandwidth, and processing requirements. There is a need to move more of this CNN processing to IoT edge and embedded devices for low-power and potentially offline, processing.

The CNN convolution layer consists of millions of multiply-accumulates (MACs), the arithmetic of which can be in fixed-point, integer or floating-point format. The CNN can operate in training mode or inference mode. During inference, the convolution layer occupies up to 90% of the computation time and energy of the CNN, convolving the input feature map (IFM) with the kernel weight data. The storage, movement of weight data, and acceleration of the convolution computation are often beyond the energy, storage and compute bounds of embedded devices.

We investigate opportunities for optimising the hardware energy efficiency, gate-level area, and execution time of the CNN convolution layer's MAC arithmetic, while maintaining inference classification accuracy of the CNN accelerator implementation. Our first contribution investigates reducing energy consumption and application-specific integrated circuit (ASIC) die area while maintaining classification accuracy of CNNs. We also investigate latency and resource efficiency when implemented in field programmable gate array (FPGA). Our second contribution focuses on decreasing software execution time of low-precision floating-point (FP) CNNs by exploiting hardware optimisation of central processing unit (CPU) vector register packing and single instruction multiple data (SIMD) bitwise instructions used in the CNN MAC.

Acknowledgements

My family: Michelle, Toby, and Zara, without whom I would have never attempted this PhD undertaking, let alone finish it. This PhD is just as much theirs as it is mine! They have had to do a lot to support me through this, for which I am forever grateful!

Dr David Gregg: The man, the professor, the legend! I always come away from meetings with David buzzing with new ideas or ways of tackling a research issue. I thoroughly enjoy working with David, bouncing wacky notions off whiteboards with him. David has always been incredibly supportive, giving guidance throughout and even working with me in the lab until 11 pm some nights. Best of all, he is a great laugh too, I mean, he still thinks I'm Australian (well, he is a stand-up comedian, and I have proof)!

Lab Mates: What a gang! Dr Andrew Anderson, Dr Aravind Vasudevan, Dr Shixiong Xu, Dr Yuan Wen, Dr Syed Asad Alam, Maria Francesca, Barbara (Basia) Barabasz, Kaveena Persand, Cormac Keane. It's been fun and productive, fostering new and innovative ideas for papers and writing a book chapter, discussing the mad politics of the world, moving labs, creating a new lab group identity in *Córais*, presenting to Science Foundation Ireland (SFI) and external professors, and more! I have loved every minute of it, and hope the collaboration continues! As Hebbian says, "cells that fire together wire together!"

SFI & Institute of Technology Carlow: This work was supported, in part, by Science Foundation Ireland grant 12/IA/1381. I extend an expression of thanks to the Institute of Technology Carlow, Carlow, Ireland, for their support.

Mom & Dad: Finally, I dedicate this to my parents, Cheryl and Jim. Wherever they are together on their continued journey in the universe, I would like to think they are proud! I love and miss you both!

JAMES GARLAND

Carlow, Ireland

31 March 2020

Contents

List of Figures

List of Tables

Acronyms

ACACES Advanced Computer Architecture and Compilation for high-performance Embedded Systems.

AI artificial intelligence.

AIG AND-inverter graph.

ANN artificial neural network.

ASIC application-specific integrated circuit.

BAT Baidu, Alibaba, Tencent.

BERT bidirectional encoder representations from transformers.

bfloat16 brain floating point 16-bit.

BIC bit clear.

BLAS basic linear algebra subprograms.

BNN binary neural network.

BRAM block RAM.

BRNN bidirectional recurrent neural network.

CIFAR Canadian Institute For Advanced Research.

CNN convolutional neural network.

CPU central processing unit.

CT computed tomography.

DARPA Defense Advanced Research Projects Agency.

DBN deep belief network.

DL deep learning.

DNN deep neural network.

DRAM dynamic RAM.

DSP digital signal processor.

DVS dynamic vision sensor.

EIE efficient inference engine.

ELU exponential linear unit.

FC fully connected.

FFT fast fourier transform.

FIFO first in first out.

FloPoCo FLOating-POint COres.

FLOPS floating-point operations per second.

FP floating-point.

FPGA field programmable gate array.

FPS Frame per Second.

FPU floating-point unit.

FSM finite-state machine.

GAN generative adversarial network.

GDPR General Data Protection Regulation.

GEMM general matrix multiply.

GFLOPS giga floating-point operations per second (FLOPS).

GMAFIA Google, Microsoft, Apple, Facebook, Intel, Amazon.

GOPS giga operations per second.

GPGPU general purpose graphics processor unit (GPU).

GPU graphics processor unit.

HDL hardware description language.

HiPEAC High Performance and Embedded Architecture and Compilation.

HLS high-level synthesis.

HOBFLOPS hardware optimized bitslice-parallel floating-point operators.

IC integrated circuit.

IET Institution of Engineering and Technology.

IFM input feature map.

ILP instruction-level parallelism.

ILSVRC ImageNet large scale visual recognition challenge.

IoT internet of things.

ISA instruction set architecture.

LSTM long short term memory.

LUT look up table.

MAC multiply-accumulate.

ML machine learning.

MLP multi-layer perceptron.

MNIST Modified National Institute of Standards and Technology.

MOSFET metal oxide semi-conductor field effect transistor.

NAN not-a-number.

NLP natural language processing.

NN neural network.

NPU network processing unit.

OFM output feature map.

OPS operations per second.

OSU Oklahoma State University.

PAS parallel accumulate and store.

PASM parallel accumulate shared MAC.

PE processing element.

PFLOPS peta FLOPS.

PRELU parametric rectified linear unit.

RAM random access memory.

RBM restricted Boltzmann machine.

RCNN Region Based CNN.

ReLU rectified linear unit.

RL reinforcement learning.

RNN recurrent neural network.

ROS robot operating system.

RTL register transfer logic.

SAT satisfiability.

SDC Synopsys design constraint.

SFI Science Foundation Ireland.

SGD stochastic gradient descent.

SHAVE streaming hybrid architecture vector engine.

SIMD single instruction multiple data.

SIPP streaming image processing pipeline.

SNARC Stochastic Neural Analog Reinforcement Calculator.

SOI silicon on insulator.

SOP sum-of-products.

SRAM static RAM.

SVE Scalable Vector Extension.

SVHN street view house numbers.

SVM support vector machine.

SWAR SIMD within a register.

TACO Transactions on Architecture and Code Optimisation.

TFLOPS tera FLOPS.

TOPS tera operations per second.

TP32 tensor float 32.

TPU tensor processing unit.

VLIW very long instruction word.

XDC Xilinx design constraint.

YOLO You Only Look Once.

Glossary

activation function A function *e.g.*, sigmoid or rectified linear unit (ReLU) that takes the weighted sum of all inputs and generates a non-linear output value.

artificial intelligence A program or model that can solve human-like tasks *e.g.*, language translation or image classification.

backpropagation The algorithm for performing gradient descent on neural networks (NNs). The output values of each neural network (NN) layer in the graph are cached in a forward pass. The partial derivatives of the errors of a layer are calculated and used to update the weight values in a backward pass through the graph.

classification accuracy The fraction of predictions that a neural network (NN) model correctly predicts.

convolution filter or weight A multi-dimensional matrix having the same number of channels as the input feature map (IFM) of the convolution layer, but has a smaller height and width dimension.

convolution operation Element-wise multiplication of the convolution filter and input feature map (IFM), followed by a summation of the resultant matrix values.

convolutional neural network A neural network (NN) which has at least one convolution layer and also usually contains activation layers, pooling layers, fully connected layers and a classification layer.

deep neural network A neural network (NN) containing multiple hidden layers.

feedforward network A network with connections in the forward direction only and do not cycle back.

gradient descent An arithmetic means to reduce loss by calculating the gradients of loss based on model's weights, and iteratively updating the training data to minimise the loss.

hidden layer A layer of a neural network (NN) between an input feature map (IFM) and output feature map (OFM) of a neural network (NN).

hyperparameter The high abstract parameters of a neural network (NN) model that can be adjusted between training epochs to train a model, *e.g.*, learning rate.

inference The process of making predictions by applying unlabelled data to a trained neural network (NN) model.

machine learning An algorithm that trains a predictive neural network (NN) model from input data.

model A neural network (NN) representation of a machine learning (ML) system that has been trained from a dataset.

neural network A model that is composed of hidden layers consisting of connected neurons followed by some non-linearity function.

neuron A node in a neural network (NN) that takes several input values and produces an output value.

node A neuron in a hidden layer of a neural network (NN).

parameter A model variable which is trained by the machine learning (ML) model.

precision The number base or arithmetic bit-precision used in the convolutional neural network (CNN).

softmax A function that normalises the input into a probability distribution consisting of probability values proportional to the exponents of the inputs. In other words, if the numbers are a large range of positive and negative values, after applying softmax the outputs will be in the interval range $(0, 1)$ so the outputs can be interpreted as probabilities.

sparse The storage of non-zero values in a neural network (NN).

uncanny valley A hypothesised relationship between the degree an object resembles a human and a human's emotional response to such an object.

1 Introduction

CONVOLUTIONAL NEURAL NETWORKS (CNNs) are a subset of machine learning (ML) which itself is a subset of artificial intelligence (AI). CNNs are very capable across a diverse range of vision and audio processing tasks. CNNs are excellent at classifying images [Krizhevsky et al. 2012], object detection of multiple objects within an image [Girshick et al. 2014a], real-time object detection in videos [Redmon et al. 2016] and pattern recognition in audio samples [Hershey et al. 2017]. CNNs require a large amount of storage, bandwidth, and computation of the host device on which it is implemented. For example, the CNN VGG-16 requires 154.7GMACs, and 138.36MB of parameters [Simonyan and Zisserman 2014b]. An increasing research focus is aimed at moving the CNN processing closer to *'edge'* IoT devices (*e.g.*, Han *et al.'s*, Deep compression work [Han et al. 2016a]) while trying to accelerate the CNN execution time and reduce energy (*e.g.*, Howard *et al.'s*, MobileNets work [Howard et al. 2017]). As researchers move the models and data sets closer to implementation on edge IoT devices, they attempt to increase efficiency and reduce the size of the associated data sets. The improved efficiency will have long term impact on the environment and prevent the 'Earth glowing red-hot,' as Stephen Hawking somewhat facetiously suggested in his address to the Tencent WE Summit, in 2017.

CNNs usually operate in one of two modes:

- *training mode* where the CNN model is *trained* to detect or recognise a pattern in data, such as a human face in an image;

- *inference* mode where the model is configured to infer or *detect* the pretrained pattern in new data, *e.g.*, detect if a human face exists in a new image not yet 'seen' by the CNN model.

This work will consider optimisations of inference operation of the CNN model in embedded systems.

1.1 Motivation

For CNNs to operate in inference mode when implemented in embedded devices, the implementation and host device requires some or all of the following:

- Reduce the execution time and memory requirements of the CNN model while maintaining the classification accuracy;
- Hardware optimisations upon which the CNN model is to run;
- Optimise the arithmetic, numeric computation and data storage precision[1]of the CNN model;
- Reduce the energy consumed by the CNN on its host device.

The acceleration of CNNs is typically split into two categories; one involves accelerating the training of the CNN models; the other category is inference acceleration.

The large energy and computational requirements of CNN inference makes it challenging to implement CNNs on low-power IoT embedded devices. A contributing factor is that 90% of the computation time and energy in a CNN model is taken by the convolution unit [Farabet et al. 2010], and more specifically, the large numbers of MAC operations required. These MACs are large in integrated circuit (IC) die area, consume a large proportion of the energy, and decrease the throughput of the CNN. While some solutions manage to optimise the CNN model and its implementation enough to run successfully on a mobile device [Howard et al. 2017], this optimisation is often at the expense of throughput, performance, and classification accuracy of the model.

[1]Note that *arithmetic precision* differs from CNN *model precision*. *Model precision* is the frequency with which a model correctly predicts a class. *Arithmetic computation and storage precision* refer to the bitwise precision of the arithmetic processing and storage medium.

1.2 Objective and Approach

The contributions of this thesis investigate optimising the target hardware architecture implementation of the MAC arithmetic at the heart of the CNN. We investigate a reduction in gate-level area and energy consumption in hardware and an improved execution time in both hardware and software.

These optimisation proposals are implemented for demonstration purposes in the MACs of the CNN inference on aspects and layers of the Deep Compression [Han et al. 2016a] and MobileNets [Howard et al. 2017] CNN models.

We ask if weight sharing, a novel compression scheme for weight data, could be optimised in embedded hardware to save memory, bus bandwidth and thus energy. Our work investigates the rearchitecting the MAC of a weight-shared CNN layer to discover if efficiencies in energy and ASIC die area and FPGA resources can be increased.

We also ask if bitslice optimisations of the MAC arithmetic, optimised with hardware tools, can increase throughput and performance of the MAC of the convolution layer of a CNN in software compiled for Arm and Intel CPUs.

1.3 Contributions

The first contribution investigates the reduction of hardware energy and IC die area while maintaining the CNNs classification accuracy in a weight-shared CNN. We call our contribution parallel accumulate shared MAC (PASM), see Chapter 3 on page 41.

We implement PASM in a weight-shared CNN convolution hardware accelerator and analyse its effectiveness. Our experiments for a weight-shared CNN layer implemented on a 45nm ASIC process at a clock speed of 1GHz show that our approach results in 35% smaller sequential logic, 66% saving in total logic gates, and 70% less total power than the equivalent standard MAC. We also show the same weight-shared-with-PASM convolutional neural network accelerator implemented in a resource-constrained FPGA. PASM consumes 99% fewer digital signal processors (DSPs) units and 28% fewer block RAMs (BRAMs) units and consumes 18% less total FPGA reported power consumption, with up to a maximum 12% increase

in latency. Therefore, the power consumption of the convolution in an embedded system can be dialed down at the expense of an increased latency of convolution.

PASM has attracted interest from industry by Thomas Guttenberger, Founder and CEO of `https://metacoder.ai/` who would like to implement PASM to reduce energy consumption.

The second contribution decreases software execution time of arbitrary-precision FP MACs contained in the convolution layer of a CNN. We exploit hardware synthesis tool optimisation and bitslicing of vector register packing and SIMD bitwise instructions for the CNN MAC. We exploit the hardware synthesiser as the C++ compiler does not perform the SIMD reduction and packing to the same efficiency. We call this contribution hardware optimized bitslice-parallel floating-point operators (HOBFLOPS), see Chapter 4 on page 71.

Our experiments show that HOBFLOPS provides a fast approach to emulating custom, low-precision FP in software. We demonstrate implementing various widths of HOBFLOPS multiplier and adder in the MAC of a CNN convolution. On Arm and Intel processors, the MAC performance in CNN convolution of HOBFLOPS, Flexfloat, and Berkeley's SoftFP are compared. HOBFLOPS outperforms Flexfloat by up to $10\times$ on Intel AVX512. HOBFLOPS offers arbitrary-precision FP with custom range and precision, *e.g.*, HOBFLOPS9, which outperforms Flexfloat 9-bit on Arm Neon by $7\times$. HOBFLOPS allows researchers to prototype different levels of custom FP precision in the arithmetic of software CNN accelerators. Furthermore, HOBFLOPS fast custom-precision FP CNNs may be valuable in cases where memory bandwidth is limited.

HOBFLOPS attracted a technology journalist to interview the lead author of the work on the Youtube channel *Next Platform*.

Having introduced the research contributions, we now present the structure of this thesis.

1.4 Thesis Structure

Chapter 2, analyses the state-of-the-art research in ML model implementations. We look at recent work from the late 1980s to today. We chart the key research

concepts over this period and discuss and evaluate their contributions and where they might lead the research contributions of this thesis.

Chapter 3, presents our first contribution, which focuses on weight sharing CNNs, which we exploit to further reduce the energy dissipated by the MAC. The weight-sharing compression scheme was chosen for the reduced energy dissipation from the memory access when fetching the compressed and quantised weight data. Weight-sharing uses a scheme of binning compressed weight values which we exploit in our proposal. We rearchitect the MAC to replace hardware multipliers in the MAC circuit with adders and selection logic, which we call *parallel accumulate shared MAC (PASM)*. Rather than computing the MAC arithmetic directly, which requires a large number of multipliers, we instead count the frequency of each weight and store the counts of each bin. A subsequent multiply phase computes the accumulated count value, sharing a single multiplier between multiple *parallel accumulate and store (PAS)* units. We significantly reduce the number of multipliers and, therefore, the IC area and energy consumption of the accelerator when implemented on ASIC. We also show that the same weight-shared-with-PASM CNN accelerator can be implemented in resource-constrained FPGAs, where the FPGA has limited numbers of DSP units to accelerate the MAC operations.

Chapter 4, presents our second main contribution, *hardware optimized bitslice-parallel floating-point operators (HOBFLOPS)*. There is value in arbitrary and low-precision FP on both embedded and high-end devices, so this work presents a method of bitwise packing and arithmetic of FP compute while maintaining or increasing performance and reducing power consumption. Our experiments show that HOBFLOPS provides a fast approach to emulating custom, low-precision FP in software, offering up to $8\times$ the performance of SoftFP16. HOBFLOPS allows researchers and hardware developers to prototype different levels of custom FP precision for use in the arithmetic of CNN accelerators, the equivalent HOBFLOPS9 of which achieves up to $6\times$ that of HOBFLOPS16. Furthermore, HOBFLOPS fast custom-precision FP CNNs in software may be valuable in cases where memory bandwidth is limited.

Chapter 5, discusses and evaluates the findings, suggests future work and concludes with some final thoughts for the research area.

Appendix A, outlines the historical background to AI. We highlight the desire for humans to automate tasks and model the physical aspects of a human to perform those tasks. We show how this desire has fuelled research and development in AI, ML and deep learning (DL), charting the major developments since World War II to present-day achievements in activities such as facial recognition, voice recognition and understanding in machines.

Appendix B, introduces CNNs, their properties and operation. We quickly bring the reader up to speed on CNNs and how they are trained and implemented for use in pattern recognition applications. We show some of the underlying mathematical operations that the CNN use to train for and perform the task of *e.g.*, image detection and recognition.

1.5 Publications

Early versions of this thesis's research appear in three refereed journal articles, one conference presentation, two poster presentations, and a co-written book chapter. At the time of writing, these works have resulted in twenty-eight citations. The PASM work has gained interest from an industry CEO. The HOBFLOPS work has garnered interest from a technology journalist.

1.5.1 Refereed Journals Articles

James Garland and David Gregg (2017). "Low Complexity Multiply Accumulate Unit for Weight-Sharing Convolutional Neural Networks". In: *IEEE Computer Architecture Letters* 16.2, pp. 132–135. ISSN: 1556-6056. DOI: 10.1109/LCA.2017.2656880

- Parts of this PASM work appear in chapter 3 on page 41.
- We presented a poster of this work at the 2017 *High Performance and Embedded Architecture and Compilation (HiPEAC) Advanced Computer Architecture and Compilation for high-performance Embedded Systems (ACACES)* summer school.

James Garland and David Gregg (2018). "Low Complexity Multiply-Accumulate Units for Convolutional Neural Networks with Weight-Sharing". In: *ACM Transactions on Architecture and Code Optimization* 15.3, 31:1–31:24. ISSN: 1544-3566. DOI: 10.1145/3233300**. URL:** http://doi.acm.org/10.1145/3233300

- We built significantly on the research of the initial PASM work above.
- A modified version of chapter 3 on page 41 of this thesis has been published in ACM Transactions on Architecture and Code Optimisation (TACO) journal.
- We presented the extended PASM work [Garland and Gregg 2018] at the HiPEAC 2019 conference in Valencia, Spain, on 23 JAN 2019 during paper track Session 12 Programming Models, Neural Networks[2]. The session Chair was Dr Luca Fanucci of the Università di Pisa[3]. The research was also presented at the student poster session at the same conference.
- Thomas Guttenberger, founder and CEO of `https://metacoder.ai/` has expressed an interest in implemented PASM in their AI servers.

1.5.2 Journal Article Submitted and Under Review

James Garland and David Gregg (2021). "HOBFLOPS for CNNs: Hardware Optimized Bitslice-Parallel Floating-Point Operations for Convolutional Neural Networks". In: ***PREPRINT (Version 1) available at Research Square.*** **DOI:** 10.21203/rs.3.rs-866039/v1**. URL:** [https://doi.org/10.21203/rs.3.rs-866039/v1

- A modified version of chapter 4 on page 71 has been submitted for review to the Springer Soft Computing for Edge-Driven Applications journal, a preprint of which Springer store at the Research Square preprint server;
- The technology journalist, Timothy Prickett Morgan, of The Next Platform[4] web site and Youtube channel[5], interviewed the lead author on Monday 27th July 2020 about our HOBFLOPS CNN work on their NextPlatformTV show.

1.5.3 Book Chapter Publications

Our initial PASM work [Garland and Gregg 2017], was published as a chapter in the 2017 HiPEAC ACACES non-peer-reviewed book.

We were invited to write a chapter to contribute to an Institution of Engineering and Technology (IET) published book in 2019.

[2] `https://www.hipeac.net/2019/valencia/#/program/paper-track`
[3] `http://www.iet.unipi.it/l.fanucci/`
[4] `https://www.nextplatform.com/`
[5] `https://www.youtube.com/channel/UC-Q65AMbMP1tGIxzSagcmHg`

Andrew Anderson et al. (2019). "Hardware and software performance in deep learning". In: *Many-Core Computing: Hardware and Software*. Ed. by Geoff V. Merrett Bashir M. Al-Hashimi. Computing. Institution of Engineering and Technology. Chap. 6, pp. 141–161. ISBN: 9781785615825. DOI: 10.1049/PBPC022E. URL: https://digital-library.theiet.org/content/books/pc/pbpc022e

1.6 Journal Articles Peer Reviewed

During the duration of the PhD, the author of this thesis peer-reviewed the following IEEE journal articles. These works and their associated journals are very much in line with this author's research.

- ***IEEE TVLSI Journal Journal:*** Sungju Ryu et al. (2018). "Feedforward-Cutset-Free Pipelined Multiply–Accumulate Unit for the Machine Learning Accelerator". In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 27.1, pp. 138–146;
- ***IEEE Access Journal Journal:*** Bo Liu et al. (2019). "An Ultra-Low Power Always-On Keyword Spotting Accelerator Using Quantized Convolutional Neural Network and Voltage-Domain Analog Switching Network-Based Approximate Computing". In: *IEEE Access* 7, pp. 186456–186469;
- ***IEEE Access Journal Journal:*** A High-Performance Multiply-Accumulate Unit by Integrating Additions and Accumulations into Partial Product Reduction Process. C. Tung and S. Huang (2020). "A High-Performance Multiply-Accumulate Unit by Integrating Additions and Accumulations Into Partial Product Reduction Process". In: *IEEE Access* 8, pp. 87367–87377;
- ***IEEE TVLSI Journal Journal:*** An Energy-Efficient Asynchronous and Reconfigurable CNN Accelerator (To be published)

Here we have briefly covered from where the research motivations stem and how they have aided the development and prototyping of the research. We will now review the taxonomy of the background literature of AI to bolster the research interests.

2 Taxonomy and Review of Neural Network Literature

2.1 Introduction

CONVOLUTIONAL NEURAL NETWORKS have become of increased interest with researchers and industry since the proposal of the AlexNet [Krizhevsky et al. 2012] deep learning (DL) neural network model. However, there are many challenges to optimising CNNs execution time, energy consumption and implementation in embedded devices, some of the challenges of which are:

- CNN Model sizes and associated training data are very large, leading to increased research in optimising the CNNs for efficient implementation and throughput in CPU, GPU, FPGA and ASIC accelerators;
- Speed of training and inference of CNNs needs to increase, but is difficult due to the large size of the CNN models and associated data;
- CNN energy efficiency needs to increase, but the large models and associated data are demanding compute and memory bandwidth that is excessive to low-power IoT edge devices.

Appendix A, outlines the historical background to AI. We highlight the desire for humans to automate tasks and to model the physical aspects of a human to perform those tasks. We show how this desire has fuelled research and development in AI, ML and DL, charting the major developments since World War II to present-day achievements in activities such as facial recognition, voice recognition and understanding in machines.

Table 2.1: Taxonomy of Related Research

	Architecture Optimisation	Algorithm Optimisation	Customising Bit Precision
Accuracy	FINN [Umuroglu et al. 2017] (**F**) Hi. Perf. FP Pipelines [De Dinechin et al. 2009] (**F**)	AlexNet [Krizhevsky et al. 2012] (**G**) Deep Compression [Han et al. 2016a] (**C,G,MG**) GoogleNet [Szegedy et al. 2015] (**C**) Granular Sparsity [Mao et al. 2017] (**P**) MobileNets [Howard et al. 2017] (**P**) ResNet [He et al. 2015] (**G**) VGGNet [Simonyan and Zisserman 2014b] (**G**)	FINN [Umuroglu et al. 2017] (**F**) Hi. Perf. FP Pipelines [De Dinechin et al. 2009] (**F**) Rethinking FP [Johnson 2018] (**A**)
Area	EIE [Han et al. 2016b] (**A**) Hi. Perf. FP Pipelines [De Dinechin et al. 2009] (**F**) Myriad 2 [Moloney et al. 2014] (**A**) Project Brainwave [Chung et al. 2018] (**F**) Rethinking FP [Johnson 2018] (**A**) tensor processing unit (TPU) [Jouppi et al. 2017] (**A**)	MobileNets [Howard et al. 2017] (**P**) Rethinking FP [Johnson 2018] (**A**)	Hi. Perf. FP Pipelines [De Dinechin et al. 2009] (**F**) Rethinking FP [Johnson 2018] (**A**)
Energy	Bismo [Umuroglu et al. 2018] (**F**) DaDianNao [Chen et al. 2015b] (**A**) DianNao [Chen et al. 2014] (**A**) EIE [Han et al. 2016b] (**A**) Eyeriss [Chen et al. 2016; Chen et al. 2019] (**A**) Myriad 2 [Moloney et al. 2014] (**A**) NeuFlow [Pham et al. 2012] (**A**) Prec'n Scalable Proc. [Moons and Verhelst 2016] (**A**) Rethinking FP [Johnson 2018] (**A**) TPU [Jouppi et al. 2017] (**A**)	Deep Compression [Han et al. 2016a] (**C,G,MG**) EIE [Han et al. 2016b] (**A**) Rethinking FP [Johnson 2018] (**A**) Green AI [Schwartz et al. 2019] (**C**)	EIE [Han et al. 2016b] (**A**) Rethinking FP [Johnson 2018] (**A**) TPU [Jouppi et al. 2017] (**A**)
Execution Time	Bismo [Umuroglu et al. 2018] (**F**) DaDianNao [Chen et al. 2015b] (**A**) DianNao [Chen et al. 2014] (**A**) Deep Learning with INT8 [Fu et al. 2016] (**F**) EIE [Han et al. 2016b] (**A**) FINN [Umuroglu et al. 2017] (**F**) Hardware Accel. CNNs [Farabet et al. 2010] (**F**) Hi. Perf. FP Pipelines [De Dinechin et al. 2009] (**F**) Myriad 2 [Moloney et al. 2014] (**A**) NeuFlow [Pham et al. 2012] (**A**) Opt. FPGA Accel. CNNs [Zhang et al. 2015] (**F**) Opt. Loop Oper'n/Dataflow [Ma et al. 2017] (**F**) Prec'n Scalable Proc. [Moons and Verhelst 2016] (**A**) Project Brainwave [Chung et al. 2018] (**F**) TPU [Jouppi et al. 2017] (**A**)	Backprop Handwritten Recog. [LeCun et al. 1989] (**C**) BinaryConnect [Courbariaux et al. 2015a] (**P**) BinaryNet [Courbariaux et al. 2016] (**G**) EIE [Han et al. 2016b] (**A**) Faster Int. Multiplication [Fürer 2007] (**P**) Grad. Learn'g of Doc. Recog. [Lecun et al. 1998] (**C**) Ltd. Numerical Precision [Gupta et al. 2015] (**F**) Opt. FPGA Accel. CNNs [Zhang et al. 2015] (**F**) Opt. Loop Oper'n/Dataflow [Ma et al. 2017] (**F**) Parallel MCMK CNN [Vasudevan et al. 2017] (**C**) Project Brainwave [Chung et al. 2018] (**F**) XNOR-Net [Rastegari et al. 2016] (**C**)	8-bit Approximation [Dettmers 2015] (**G**) Bit-slice FP [Xu and Gregg 2017] (**C**) Deep Compression [Han et al. 2016a] (**C,G,MG**) Deep Learning with INT8 [Fu et al. 2016] (**F**) Fixed-point Quantisation [Lin et al. 2016] (**P**) Flytes [Anderson et al. 2017] (**C**) Hi. Perf. FP Pipelines [De Dinechin et al. 2009] (**F**) Ltd. Numerical Precison [Gupta et al. 2015] (**F**) MobileNets [Howard et al. 2017] (**P**) Project Brainwave [Chung et al. 2018] (**F**) TPU [Jouppi et al. 2017] (**A**)
Storage	EIE [Han et al. 2016b] (**A**) FINN [Umuroglu et al. 2017] (**F**)	BinaryConnect [Courbariaux et al. 2015a] (**P**) BinaryNet [Courbariaux et al. 2016] (**G**) Distilling Knowledge [Hinton et al. 2015] (**P**) Fixed-point Quantisation [Lin et al. 2016] (**P**) SqueezeNet [Iandola et al. 2016] (**P**)	Deep Compression [Han et al. 2016a] (**C,G,MG**) SqueezeNet [Iandola et al. 2016] (**P**) XNOR-Net [Rastegari et al. 2016] (**C**)

There are many ways of tackling the above challenges. This chapter will look at the current research literature and discuss how researchers are addressing these challenges. The review is organised using the taxonomy shown in Table 2.1. The hardware and software levels of abstraction are the primary axis (shown in the headings at the top) of Table 2.1.

The orthogonal axis (headings on the left of Table 2.1) consists of the major research themes of the works, ideas and techniques, focusing on optimising the target hardware, optimising the algorithm or customising the bit-precision:

- **Accuracy**: Research that focuses on optimising the number representation accuracy or CNN prediction accuracy;
- **Area**: How researchers optimise the utilised area of IC or memory footprint of the CNN;
- **Energy**: Optimising the energy dissipated within the CNN, with a focus on inference;
- **Execution Time**: Research proposals put forward to optimise the speed, throughput or latency of the CNN. Much focus on execution time, both within software and hardware has been the driver of a great deal of CNN research;
- **Storage**: The quantity of storage required by the CNN due to the optimisation strategies. The storage and movement of the weights and parameters data will increasingly move toward the memory elements.

In brackets after each piece of research, a letter appears in bold. These letters indicate the implementation technology which may impact the energy consumption, throughput and accuracy of the CNN:

- **A** means the research work implements the researcher's proposal in an ASIC;
- **F** shows the target is FPGA implementation;
- **G** implements the proposal in general purpose GPU (GPGPU);
- **MG** targets a mobile GPU for the proposals implementation;
- **C** indicates a CPU implementation;
- **P** suggests the technique doesn't state how the proposal is implemented; it is presumed the contribution is implemented in software in a CPU.

We categorise the focus or investigation of each research area to determine under which primary and orthogonal axis to place the work. Entries may appear more

than once where the entries support either multiple research themes or multiple levels of abstraction. The levels of abstraction form the sections of this chapter. Each research theme and implementation technology is addressed within the levels of abstraction sections.

While addressing the above challenges researchers implement various optimisation strategies, where applicable, in their hardware and software implementations. There are several ways to optimise for accuracy while focusing on execution time and energy dissipation, the popular methods of which are:

1. Optimise the CNN architecture *e.g.*,
   (a) Create larger deep CNN models for improved classification accuracy (see Table A.2);
   (b) Compress, prune or change the sparsity of the weight and/or input feature map (IFM) data;
   (c) Create a new layer type with few weights;
   (d) Create alternative convolution methods;
   (e) Produce a software library of functions, such as basic linear algebra subprograms (BLAS);
2. Optimise the number precision and operators of the CNN *e.g.*,
   (a) Reduce the precision of the number format and arithmetic used in the algebra of a CNN;
   (b) Optimise arithmetic operators;
   (c) Use other techniques such as bit slicing;
3. Create custom hardware accelerators *e.g.*,
   (a) Create a specific DSP vector processor;
   (b) Create specific FPGA and ASIC accelerators;
4. Focus specifically on energy conservation *e.g.*,
   (a) Create space and time efficient models to fit on low-power devices;

(b) Improve data efficiency by reducing data movement and reducing the overall data requirements.

We shall investigate how the works of Table 2.1 use these optimisation strategies.

2.2 Architecture Optimisations

The next subsections will examine the architectural optimisations of the CNN model (see column 1 of Table 2.1) mainly from an implementation perspective. However, as we will see, the optimisations of the implementation of the CNN model often go hand-in-hand with high-level model optimisations. We will highlight various sub-categories with capitalised sub-headings, corresponding to the above-itemised challenges.

2.2.1 Architecture Optimisation - Accuracy

Researchers strive to maintain or increase the CNN classification accuracy while optimising CNN models for the target architecture such as an FPGA.

2. OPTIMISE THE NUMBER PRECISION AND OPERATORS OF THE CNN:

2(a). REDUCE PRECISION OF NUMBER FORMAT AND ARITHMETIC:

A popular optimisation method is low-level reduction in precision of the number format and arithmetic performed by the CNN model. For example, reducing weights and arithmetic to their binary representations. Umuroglu *et al.*, [2016] propose FINN, a binary neural network (BNN) framework for building fast and flexible FPGA accelerators. Using the Canadian Institute For Advanced Research (CIFAR)-10 [Krizhevsky and Hinton 2009] and street view house numbers (SVHN) [Netzer et al. 2011] datasets, Umuroglu *et al.*, show 21906 image classifications per second with $283\mu s$ latency and 80.1% and 94.9% classification accuracy respectively, the fastest classification in FPGA at the time of their publication. However, training the CNN with reduced binary representation weights is time-, compute- and energy-intensive. Figure 1 of [Courbariaux et al. 2016] shows BNNs are at least $10\times$ slower to train, but exhibit accuracy close to that of 32-bit float deep neural networks (DNNs).

Design of FPGA and ASIC hardware accelerators at the register transfer logic (RTL) level with reduced bitwise precision, can be laborious and prone to errors. De Dinechin *et al.'s*, [2009; 2011] propose automated methods of producing VHDL components with custom precision that can be used in CNNs. We exploit this type of automation in our HOBFLOPS proposal (section 4.1).

2.2.2 Architecture Optimisation - Area

Today, CNNs are often very large such as ResNet [He et al. 2015]. The model size has led researchers to optimise the overall area of the accelerator designs. Researchers often rearchitect the CNN to implement it in a DSP/vector processor or custom ASIC and FPGA based CNN accelerators.

2. OPTIMISE THE NUMBER PRECISION AND OPERATORS OF THE CNN:

2(a). REDUCE PRECISION OF NUMBER FORMAT AND ARITHMETIC:

Designing dedicated hardware for FPGA and ASIC allows designers to produce different number representations and data types. Alternative number representations reduce the data storage requirements, thus allowing a smaller gate-level area of accelerator design to be produced.

Johnson [2018] proposes an alternative floating-point representation and a hybrid log multiply/linear add function, Kulisch accumulation, and tapered encodings from Gustafson's posit format for accelerators. Johnson shows that a 16-bit log float multiply-add is $0.68\times$ the IC die area compared with an IEEE-754 float16 fused multiply-add while maintaining the same significand precision and dynamic range.

Microsoft propose the alternative MS-FP8 and MS-FP8 FP representations [Chung et al. 2018], a highly quantised version of IEEE FP-754 standard, allowing tight implementation in FPGA DSP units.

3. CREATE CUSTOM HARDWARE ACCELERATORS:

3(a). CREATE SPECIFIC DSP/VECTOR PROCESSOR:

The streaming hybrid architecture vector engine (SHAVE) v3.0 processor in the Myriad 2 [Moloney et al. 2014] is a 128-bit SIMD 12-very long instruction word (VLIW) vector processor which can support various bit widths of integer and FP representations. Within the Myriad 2, there are 12 SHAVE processors connected to their streaming image processing pipeline (SIPP) computational imaging hardware accelerators via a crossbar. Myriad 2 employs optimised scratch pads, caches,

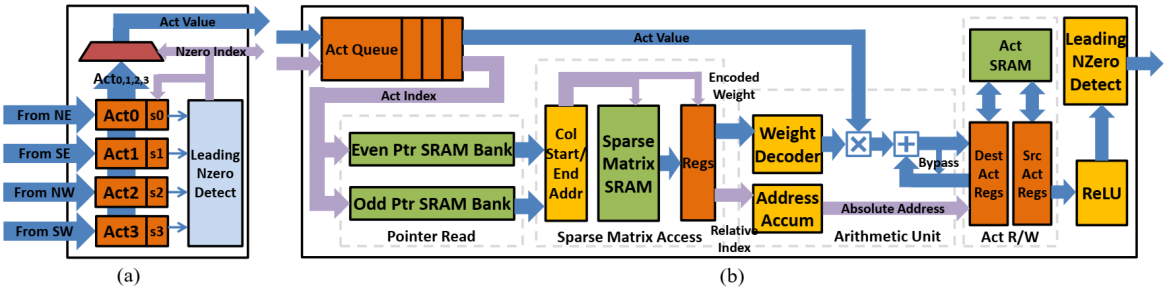


Figure 2-1: **The architecture of the Leading Non-zero Detection Node (a) and architecture of Processing Element (b)** [Han et al. 2016b].

instruction-level parallelism (ILP), first in first outs (FIFOs), and other queuing mechanisms. Myriad 2 produces an accelerator that has very competitive giga FLOPS (GFLOPS)/W compared with the Nvidia's Tegra K1 GPU accelerator but with a very small IC die size. Myriad 2 is often used in embedded low-power applications, such as DL vision systems on the DJI Spark and other drones [Movidius 2017].

3(b). CREATE SPECIFIC FPGA and ASIC ACCLERATOR:

De Dinechin *et al.*, [2009; 2011] show that they can control the post synthesis gate-level area of both of their 32-, and 64-bit RTL accelerators with their automation mentioned above.

Han *et al.*, [2016b] evaluate a pruned and compressed (via weight-sharing) inference engine implemented in both a TSMC 45nm and 28nm CMOS process ASICs. Their efficient inference engine (EIE) hardware accelerator can store all weights in on-chip static RAM (SRAM). They show that compared with DaDianNao [Chen et al. 2015b] of Chen Y. *et al.*, EIE achieves $3\times$ smaller area. Han *et al.*, layout a processing element (PE) and approximate the area of a PE to be $638,024\mu\ m^2$. Figure 2-1 demonstrates the level of rearchitecting the CNN undergoes to reduce the area of the CNN model and associated data. Their optimisations show no loss in accuracy.

Google's TPU [Jouppi et al. 2017] area, when implemented in ASIC silicon measures around $331mm^2$ (see Figure 2-2 for a block diagram). This compares favourably to the competing Intel Haswell CPU and Nvideo K80 GPU which are roughly double the IC die area.

2.2.3 Architecture Optimisation - Energy

Often researchers target FPGA and ASIC architectures as they generally dissipate lower energy than CPU or GPU implementations of the CNN accelerator.

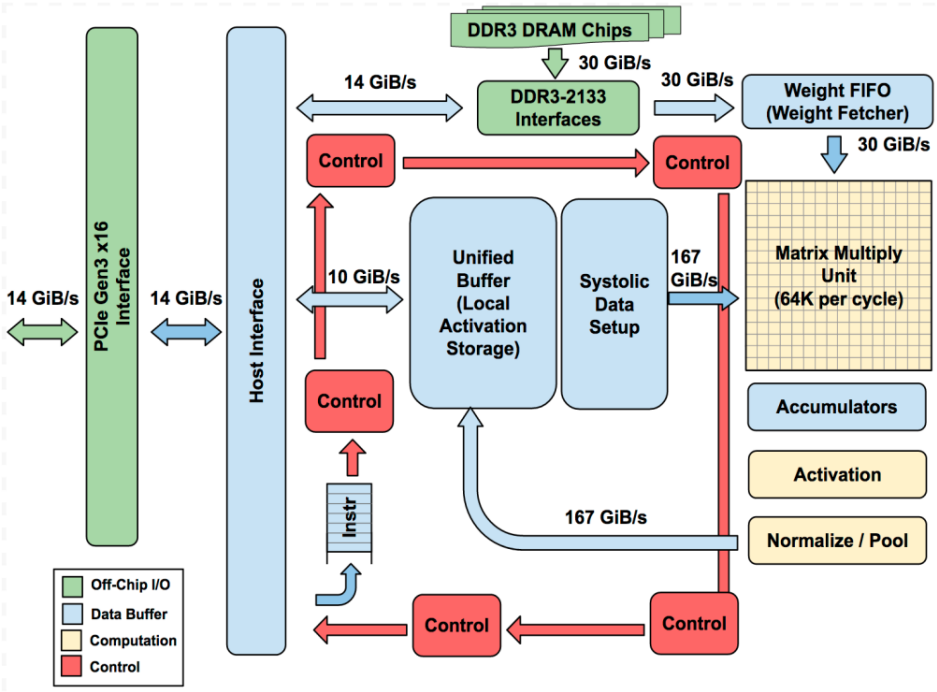


Figure 2-2: **TPU Block Diagram** [Jouppi et al. 2017].

1. OPTIMISE THE CNN ARCHITECTURE:

1(b). COMPRESSION, PRUNING, SPARSITY:

Moons *et al.*, built on their Energy Efficient ConvNets work [Moons et al. 2016] to produce the Precision Scalable Processor [Moons and Verhelst 2016], a CNN ASIC that is a custom fully C-programmable processor. Their proposal applies quantisation schemes down to layer granularity, applies precision scaling to their accelerator circuits and sparsity to reduce the computations needed due to the zero-valued parameters of the precision scaling. Moons *et al.*, claim the processor consumes 25-288mW at 204 MHz.

2. OPTIMISE THE NUMBER PRECISION AND OPERATORS OF THE CNN:

2(a). REDUCE PRECISION OF NUMBER FORMAT AND ARITHMETIC:

Google's TPU accelerator [Jouppi et al. 2017] uses brain floating point 16-bit (bfloat16) in the systolic arrays to accelerate the matrix multiplication operations. They use 32-bit IEEE FP for accumulation and claim their TPU has a tera operations per second (TOPS)/Watt about 30X – 80X higher than that of contemporary GPUs or CPUs. As Google implement the TPU circuits in an ASIC, they intrinsically produce a lower energy CNN accelerator and claim around 40W with a peak 92TOPS, however Google does not publish exact comparisons of their energy reduction.

Bismo FPGA accelerator [Umuroglu et al. 2018] build hardware cost models of CNNs and show an energy efficiency of up to 1.4 TOPS/W on a PYNQ-Z1 board, impressive for a small, circa US $200 consumer FPGA development board. Bismo uses BNNs so inference is performed at the meagre energy budget of the PYNQ-Z1.

Changing the FP number representation, Johnson [2018] work mentioned earlier shows that in 16 bits, their log float multiply-add is $0.59\times$ the energy of IEEE-754 float16 fused multiply-add, in a 28nm ASIC process geometry.

3. CREATE CUSTOM HARDWARE ACCELERATORS:

3(a). CREATE SPECIFIC DSP/VECTOR PROCESSOR:

Independent analysis of the Myriad [Moloney et al. 2014] vector processor performance [Ionica and Gregg 2015] show that Myriad 1 exhibits a performance/watt ratio of 23.17 GFLOPS/W while Movidius claim a 500mW operation.

3(b). CREATE SPECIFIC FPGA and ASIC ACCELERATOR:

Custom accelerators are often more expensive initially to produce than vector processors but allow for reduced energy consumption, smaller devices, and cheaper mass production. Hardware can have reduced overhead of fetch/decode/execute of instructions and reduced complex pipelines with ILP, by either reducing the instruction set architecture (ISA) or removing the ISA completely and implementing a finite-state machine (FSM) in RTL. NeuFlow [Pham et al. 2012] does exactly this by implementing Farabet's NeuFlow [2011] on an IBM 45 nm silicon on insulator (SOI) process. The implementation accelerates large numbers of convolutions and matrix-to-matrix operations, thus reducing data movement due to the lower bit-representation. NeuFlow can deliver up to 320 giga operations per second (GOPS) with an average power consumption of 0.6W.

By optimising the locality and bandwidth of the required weight-data memory accesses, DianNao, [Chen et al. 2014] implements large CNNs in a 65nm process ASIC. Chen T. *et al.*, rearchitect their ASIC to perform 496 parallel 16-bit fixed-point operations, reducing the energy by $21.08\times$ than that of a comparable 128-bit SIMD processor core clocked at 2GHz.

Later Chen Y. *et al.*, propose DaDianNao [2015b] which shrinks the process geometry of their ASIC accelerator to 28nm. Storage of the enormous weight data is tackled by spreading the storage of all weights across the multiple accelerator nodes'

on-chip dynamic RAM (DRAM) or SRAM, thus requiring no main memory. DaDianNao outperforms a single GPU reducing energy consumption by up to $150.31\times$ when using 64 nodes, taking 6.12W memory power and 15.97W total power. However, DaDianNao cannot exploit sparsity or weight sharing of the weights and activations as the exploitation would need to expand the network to dense form before an operation, which would increase energy consumption.

The pruning, compression and weight-sharing scheme of EIE [Han et al. 2016b] highlighted above, demonstrate a the compressed number of MAC operations. The optimisation achieves $19\times$ better energy efficiency than DaDianNao.

The Eyeriss [Chen et al. 2017; Chen et al. 2019] ASIC accelerators exploit the row stationary dataflow with 168 processing elements and 16-bit fixed-point integer operations, thus reducing data movement and energy. Chen *et al.*, claim that the row stationary dataflow reconfigures the computation mapping of a given shape, maximally reusing data locally to reduce expensive data movement, such as DRAM accesses and thus increasing energy efficiency. Eyeriss 2 tiles the MAC spatially across the PEs through any of the dimensions of the layers. The layers can vary in shape and size so to deal with these differences, Eyeriss 2 uses a hierarchical mesh that adapts to these variations of kernels thus improving the computation resources to perform $2.5\times$ more energy efficiency than the predecessor Eyeriss v1 when running MobileNets.

2.2.4 Architecture Optimisation - Execution Time

Different researchers report different styles of metrics, *e.g.*, frames per second or TOPS, or the researchers implement different styles of an accelerator. However, all contributions give some form of metric that often shows an improvement impacted by their architectural change or form of implementation.

2. OPTIMISE THE NUMBER PRECISION AND OPERATORS OF THE CNN:

2(a). REDUCE PRECISION OF NUMBER FORMAT AND ARITHMETIC:

Han *et al.*, apply their deep compression optimisation techniques (see Figure 2-3) introduced above to VGG-16 CNN and find the model runs $3\times$ to $4\times$ faster on a mobile GPU with no loss in accuracy [Han et al. 2016a]. Their EIE work [Han et al. 2016b] achieves $2.9\times$ better throughput than DaDianNao [Chen et al. 2015b],

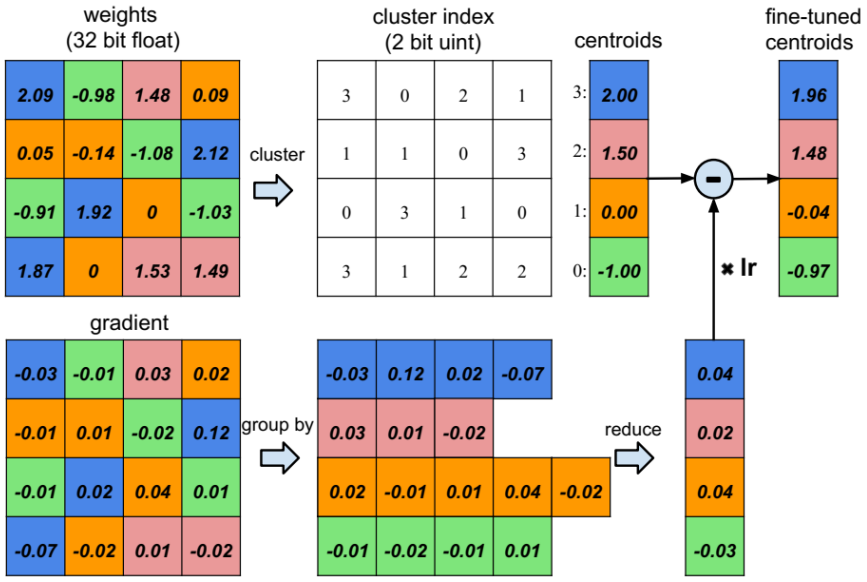


Figure 2-3: **Weight sharing by scalar quantisation (top) and centroids fine-tuning (bottom).** [Han et al. 2016a].

impressive considering EIE is implemented on the older 45nm process technology and DaDianNao is implemented on the newer, more efficient 28nm process.

Moons *et al.'s*, Precision Scalable Processor [Moons and Verhelst 2016] demonstrates yields of 0.3-2.6TOPS/W, which is $3.9\times$ higher than that of Eyeriss [Chen et al. 2016].

Researchers in industry often settled on an 8-bit architecture for the CNN. 8-bit architectures are less time consuming to train and implement than a BNN with most of the benefits of BNNs and most of the classification accuracy of FP accelerators. INT8 [Fu et al. 2016] changes the bit level type and precision to 8-bit integers and achieves $1.75\times$ peak solution-level performance deep learning operations per second (OPS) compared to other FPGAs with the same resources.

Later in 2018, Microsoft's MSFP8 and MSFP9 [Chung et al. 2018] proposal mentioned above again changes the number format and datatype to their FP 8-bit and produce 720GOPS/W on an Intel Stratix 10 280.

3. CREATE CUSTOM HARDWARE ACCELERATORS:

3(a). CREATE SPECIFIC DSP/VECTOR PROCESSOR:

Myriad 1 [Moloney et al. 2014] has a performance of around 8.11GFLOPS, which Movidius increased to 200GFLOPS in Myriad 2, and Myriad X boasts 4 TOPS of operation.

3(b). CREATE SPECIFIC FPGA and ASIC ACCELERATOR:

De Dinechin *et al.*, [2009] showcase their custom accelerator on a Xilinx Virtex-4 part, showing performance can increase from 10 cycles to 29 cycles for 32-bit FP to 10 to 46 cycles for 64-bit FP when implementing the FP Collision operator on the FPGA.

NeuFlow [Farabet et al. 2011; Pham et al. 2012] implemented on an IBM 45 nm SOI process can deliver up to 320GOPS.

Ma *et al.*, [2017] propose a CNN accelerator demonstrated on an Intel Altera Arria 10 GX 1150 FPGA. Ma *et al.*, implement a VGG-16 CNN network which achieves 645.25 GOPS of throughput and 47.97ms of latency, a $> 3.2\times$ enhancement compared to other FPGA implementations, and very roughly $10\times$ that of the accelerator of Zhang *et al.*, [2015].

Bismo FPGA accelerator [Umuroglu et al. 2018] improve upon their FINN accelerator [2017] to build hardware cost models of CNNs in a bit-serial precision and show a peak performance of 6.5TOPS on a PYNQ-Z1 board, orders of magnitude greater than the above predecessors.

Several ASIC implementations change the number representation to fixed-point.

DianNao (Figure 12 of [Chen et al. 2014]) claim that there is a minimal classification accuracy trade-off when comparing the training of Modified National Institute of Standards and Technology (MNIST) using fixed point and FP representations. 32-bit FP has an error rate of *0.0311* while 16-bit fixed-point has an error rate of *0.0337*. However, the storage, bandwidth, and compute needed for fixed-point is significantly smaller than with FP. Chen *et al.*, applies tiling of their loop nest for data-locality optimisation of the convolution, pooling and classifier layers. DianNao is implemented on a 65nm process ASIC and can perform 496 parallel 16-bit fixed-point operations in 1.02ns. Chen *et al.*, claim the accelerator to be $117.87\times$ faster than the comparable 128-bit SIMD processor core clocked at 2GHz.

DaDianNao [2015b], implemented on a 28nm ASIC compared to DianNao's 65nm implementation, outperforms a single GPU by up to $450.65\times$ when using 64 nodes, a massive increase in performance.

The first generation of the TPU [Jouppi et al. 2017] was an 8-bit integer matrix multiplication accelerator and tended to be used in inference only. Their second-generation moved from fixed-point to FP support, so it could accelerate training

and have increased classification accuracy. Google significantly improved the memory bandwidth to 600GB/s and performance of 45tera FLOPS (TFLOPS). Google arranged the TPUs into 4-chip modules to raise the performance to 180TFLOPS. Google goes further to cascade 64 modules increasing performance to 11.5peta FLOPS (PFLOPS). Their third-generation TPU claims to be twice as powerful as the second-generation, suggesting up to 23PFLOPS of performance. Google achieve this performance by deploying the TPUs in pods with four times the number of TPU ICs as TPU 2 showing an 8-fold increase in FLOPS.

Google reduced the functionality of their cloud-TPU to deliver their edge TPU, called *Coral*. Coral is capable of 4TOPS while dissipating 2W. However, Coral only supports 8-bit operations, so it is targeted at inference only using Google's Tensorflow Lite framework. Users have to train the CNNs on a Tensorflow quantisation-aware system [Abadi et al. 2016], such as their cloud-based TPU offering.

Farabet *et al.*, [2010] show the first comparisons of performance of a CNN implemented on CPU, FPGA and ASIC. The CPU is an Intel Core 2 Duo, the FPGA is a Xilinx Virtex 4 device and the ASIC is performed in simulation only and report frames/second as their metric. The researchers compare CPU, FPGA and ASIC implementations of their CNN and show their FPGA implementation is approximate an order of magnitude faster than a CPU implementation and an order of magnitude slower than a comparable GPU implementation.

The FPGA implementation by Zhang *et al.*, [2015] on a Xilinx VC707 development board containing a Virtex 7 XC7VX485T part, designed 5 years after Farabet's, demonstrate 61.62GFLOPS performance for their CNN accelerator.

COMPARISON OF DIFFERENT HARDWARE AND SOFTWARE PLATFORMS:

Due to the inconsistent reporting of metrics of neural networks, Velasco-Montero *et al.*, [2019] attempt to tackle the inconsistent reporting issue by running CNN inference on the CNN frameworks Caffe, OpenCV, TensorFlow, and Caffe2. The researchers execute the frameworks on an Arm Cortex-A53 multi-core processor platform, in this case, the Raspberry Pi 3 Model B. Using these frameworks, the researchers implement GoogLeNet, ResNet-50 and SqueezeNet-v1.1 and capture CPU utilisation, average throughput Frame per Second (FPS), instructions executed

per second, data memory accesses, including cache loads and cache misses and branch prediction. There is no clear winner, but the results show that if *e.g.*, high throughput on GoogLeNet is required then implement it with the TensorFlow framework, whereas Squeezenet should be implemented with OpenCV for a 3.5-4FPS throughput. Velasco-Montero *et al.*, supply results to allow for a user to select the best framework for the CNN of choice when implemented on a Raspberry Pi 3.

2.2.5 Architecture Optimisation - Storage

In inference mode, as the pretrained weight data is known, it can usually be pruned, compressed or represented with a different number format to reduce its size and therefore storage needs.

2. OPTIMISE THE NUMBER PRECISION AND OPERATORS OF THE CNN:

2(a). REDUCE PRECISION OF NUMBER FORMAT AND ARITHMETIC:

Deep Compression [2016a] achieves compression factors of $35\times$ for AlexNet and $49\times$ for VGG-16. The compression subsequently reduces AlexNet from 240MB to 6.9MB and VGG-16 from 552MB down to 11.3MB. The researchers also show that even though the fully connected layers dominate the model size by 90%, these layers compress the most by up to 96% of weights pruned in VGG-16 CNN.

The FINN accelerator [2017] shows that their BRAM utilisation is very efficient, mainly due to the use of BNN.

ARCHITECTURAL OPTIMISATIONS CONCLUSION:

Architectural optimisations have proven fruitful for researchers in ASIC, FPGA implementation. Researchers typically choose their preferred research area, such as execution time, but not without affecting to one of the other orthogonal areas. Researchers select their preferred hardware implementation, such as FPGA. Some researchers move designs from FPGA to an ASIC, thus saving more energy and gaining greater performance. New architectures are appearing such as the Greenwaves RISC-V processor based GAP-8 [Flamand et al. 2018] processor. GAP-8 is gar-

nering increasing research focus, as this is typically cheaper than competing Arm-based processors due to the open-source nature.

2.3 Algorithm Optimisation

Researchers optimise CNNs to lower test error and increase classification accuracy for a given dataset, however, this is often at the expense of compute and memory bandwidth. This section will highlight works (see column 2 of Table 2.1) that attempt an algorithmic optimisation to improve compute and bandwidth, regardless of the hardware or software platform in which it is implemented. We will highlight various sub-categories with capitalised sub-headings.

2.3.1 Algorithmic Optimisation - Accuracy

As demonstrated by the ImageNet large scale visual recognition challenge (ILSVRC) winners, Table A.2, classification accuracy was the driving force of research in CNNs and still remains a driving influence.

1. OPTIMISE THE CNN ARCHITECTURE:

1(a). CREATE LARGER DEEP CNN MODELS:

Large neural network models of many layers are often used for increased classification accuracy *e.g.*, AlexNet [Krizhevsky et al. 2012] and VGG-16 [Simonyan and Zisserman 2014b]. However, training time is impacted. To speed up model training, AlexNet distributes the load over two GPUs, as shown in Figure A-11. Krizhevsky *et al.*, achieve the ILSVRC winning Top-5 test error rate (see Table A.2) by by employing the regularisation method called "dropout" proposed previously by Srivastava *et al.*, [2013], to reduce overfitting and, reduce the data movement slightly within the model. While the load distribution of training decreased the training time, it increases the overall size and energy consumption of the network, making training of AlexNet in a mobile device infeasible.

In comparison, when pretrained, the VGG-16 has 552MB of trained weight data, and sixteen convolution layers in VGG-16. The greater depth of layers enabled Simonyan *et al.*, to achieve a top-5 error rate, far surpassing AlexNet, but again preventing implementation of the model and associated data on a mobile device.

As shown in Table A.2, over the five years of the ILSVRC competition, the proposed models' number of layers, MAC count and the number of parameters rose considerably. However, in general, the increase of these metrics led to the top-5 and top-1 error rate gradually falling, surpassing the top-5 error rate of humans by 2016 [He et al. 2015]. Although not shown on Table A.2 both top-5 and top-1 error rates have continued to decrease. FixEfficientNet [Touvron et al. 2020] boasts top-5 and top-1 error rates of 88.5% and 98.7% respectively with 480 million parameters.

1(b). COMPRESSION, PRUNING, SPARSITY:

An alternative to large CNNs is to make the CNNs *"as simple as possible but no simpler"*, to quote Einstein. Researchers focus on making the CNN more sparse by reducing the network weights, the most significant component of a CNN to store and compute. There are several methods to perform CNN reduction; pruning, quantisation, sparsity and reduced arithmetic precision are some of the effective methods.

Han *et al.*, [2016a; 2016b] found that in a fully trained CNN, similar weight values occur many times. They proposed scalar quantisation of the weight data by clustering around centroids to dictionary compress the weights into bins. They found that between tens to hundreds of weight values were sufficient in network inference while maintaining the high classification accuracy rate. They encode the compressed weights with an index that specifies which of the shared weights should be used. This form of pruning and compression reduces the number of connections by $9\times$ to $13\times$, with *no impact on classification accuracy*. We exploit this work for our PASM proposal of chapter 3 on page 41.

Mao *et al.*, demonstrate different granular levels of sparsity and how the different levels affect efficiency and classification accuracy. They show the surprising outcome that they can increase the Top-5 classification accuracy, albeit slightly with their fine-grained pruning approach (*i.e.*, prune individual weights) [Mao et al. 2017]. Their proposal is an architectural optimisation as they do not change the model algorithm. They only change the architecture of the model with different styles and levels of sparsity.

1(c). CREATE NEW LAYER TYPE WITH FEW WEIGHTS:

As image recognition has surpassed human performance [Russakovsky et al. 2015], research effort is being applied to reducing the size of the network, weights,

and parameters to be stored. Google was one of the first to drive down the number of parameters and MACs required with their Inception model (see Table A.2).

Proposed by Howard *et al.*, MobileNets [2017] is a different take on the convolution schemes of other CNNs such as Inception (or GoogleNet as it has become known). Regular convolution filter or weights are replaced with depthwise convolution followed by pointwise convolution filters, a combination known as depthwise separable convolution, in a 30-layer configuration. The researchers compare a full convolution MobileNets and a depthwise separable convolution MobileNets. The depthwise separable convolution loses circa 1% classification accuracy but has approximately 10% of the multiply-adds. When comparing the accuracy of the Stanford Dogs training set [Khosla et al. 2011], with Google's Inception V3, which has 84% top-1 accuracy, MobileNets exhibits 83.3% accuracy but lower computational complexity and storage than Inception needs (see Table 10 of [Howard et al. 2017]).

2.3.2 Algorithmic Optimisation - Area

The area of CNNs, *e.g.*, IC floor plan area or memory footprint, have to be reduced for a CNN to be implemented on a mobile device.

1. OPTIMISE THE CNN ARCHITECTURE:

1(c). CREATE NEW LAYER TYPE WITH FEW WEIGHTS:

The depthwise separable convolution of MobileNets [2017] vastly reduces the size of the parameters to around 14% of that required by a comparable GoogleNet. This optimisation only loses 0.7% classification accuracy, but the model requires only 10% of the parameters of Inception requires, allowing for implementation in small embedded systems at the edge.

2. OPTIMISE THE NUMBER PRECISION AND OPERATORS OF THE CNN:

2(a). REDUCE PRECISION OF NUMBER FORMAT AND ARITHMETIC:

Johnson [2018], discussed earlier, suggests "Rethinking floating-point", which they claim is a drop-in replacement for all FP32 maths and parameters (note: only

round-to-nearest-even is supported). The replacement is only 8-bits and so is only $0.68\times$ the area of an equivalent IEEE-754 FP16 fused multiply-add.

2.3.3 Algorithmic Optimisation - Energy

Larger CNN models generally mean greater memory references and associated bus bandwidth and energy dissipation. Another challenge with CNNs is the irregular computation pattern.

1. OPTIMISE THE CNN ARCHITECTURE:

1(b). COMPRESSION, PRUNING, SPARSITY:

The sparsity of weight data of Deep Compression [2016a] reduces the energy of data movement. Weight-sharing leads to $19\times$ better energy efficiency compared to DaDianNao, with a slight increase in classification accuracy. Han *et al.*, further tackle the issues of irregular computational patterns, [2016b]. The researchers implement a sparse matrix which exhibits 90% static sparsity in the weight data, $10\times$ less computation and $5\times$ less memory footprint. Their sparse vectors perform with 70% dynamic sparsity in the activation data and $3\times$ less computation all driving down the energy consumption. They demonstrate that their weight-shared values can be stored on-chip consuming 5pJ per access rather than in off-chip DRAM that consumed 640pJ per access when implemented on a CPU/GPU system.

2. OPTIMISE THE NUMBER PRECISION AND OPERATORS OF THE CNN:

2(a). REDUCE PRECISION OF NUMBER FORMAT AND ARITHMETIC:

EIE [2016b] uses weight sharing of 4-bit weights and exhibits $8\times$ less memory footprint. The result of the irregular patterns mentioned above means the data can fit in SRAM and consume $120\times$ less energy than the equivalent implementation of the network in DRAM. Furthermore, savings are multiplicative.

Reducing the overall computational complexity, Johnson [2018] mentioned above shows that number representation of the network leads to an energy reduction. The multiply-add functionality of the CNN is reduced by $0.59\times$ that of an IEEE-754 FP16 fused multiply-add with no loss in exponent dynamic range, and maintaining the significand precision.

4. ENERGY CONSERVATION:

Schwartz *et al.*, of the Allen Institute, published some work [2019] that suggest the following three points:

1. Green AI is good for the environmental aspects, but Schwartz *et al.*, also suggest that universities should stop getting involved with "brute-force-AI". The Baidu, Alibaba, Tencent (BAT) and Google, Microsoft, Apple, Facebook, Intel, Amazon (GMAFIA) have enough compute and storage resources toward solving the brute-force problems. Instead, Schwartz *et al.*, suggest universities should look at how to best optimise an AI problem for low floating-point operations and low use of utility resources such as local compute and storage;

2. Schwartz *et al.*, suggest researchers do not use a training set with *e.g.*, every face on the planet. The face data could violate various privacy laws such as General Data Protection Regulation (GDPR) and lead to scaling problems;

3. Be more *"green"* by trying to define a problem where you minimise the "black-box" area of deep learning that cannot be described. There will always be a black-box, but researchers should make the black-box as small as possible so that the internals of the network can be understood and optimised.

2.3.4 Algorithmic Optimisation - Execution Time

While researchers strive to move CNNs closer to the edge, execution time is often negatively affected. Here we will look at how researchers tackle execution time.

1. OPTIMISE THE CNN ARCHITECTURE:

1(a). CREATE LARGER DEEP CNN MODELS:

LeNet [LeCun et al. 1989; Lecun et al. 1998] CNNs are examples of late 20^{th} century CNNs used in practice to deal with the performance of large amounts of low-level information. While LeNet has many connections, it has few parameters resulting in short training time. The final trained network and weights are small enough to fit onto commercial DSP hardware of the time and can process more than ten digits per second.

1(d). CREATE ALTERNATIVE CONVOLUTION METHODS:

Vasudevan *et al.*, [2017] propose CNN convolution using general matrix multiply (GEMM) that avoids the k^2 growth in the size of the input data. The avoidance of the growth increases the size of an intermediate matrix. The researchers produce k^2 partial results in the result of the matrix multiplication, unlike Cho and Brand [2017],

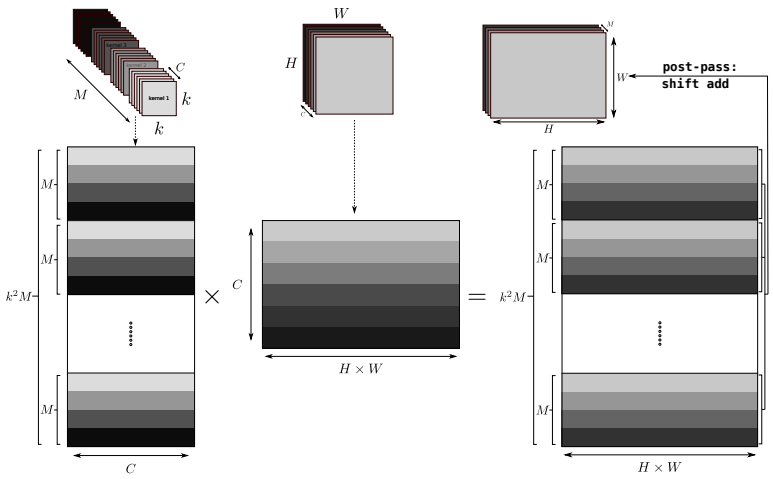


Figure 2-4: **The *kern2row* method of performing convolution using matrix multiplication.** [Vasudevan et al. 2017].

who compute the outputs in sections. Vasudevan *et al.*, call their contribution the *kern2row* method, see Figure 2-4. The output of the matrix multiplication is a matrix that is k^2 times larger than the output. They sum the partial results within the large matrix to produce the correct, smaller output matrix.

2. OPTIMISE THE NUMBER PRECISION AND OPERATORS OF THE CNN:

2(a). REDUCE PRECISION OF NUMBER FORMAT AND ARITHMETIC:

As mentioned earlier, Project Brainwave [Chung et al. 2018] produces an 8-bit (1-sign bit, 5-exponent bits and 2-mantissa bits) and 9-bit (1-sign bit, 5-exponent bits and 3-mantissa bits) FP arithmetic that Microsoft exploits in a quantised CNN. MS-FP8 and MS-FP9 offer different representative range and precision to standard FP8 or INT8. When compared to 8-bit integer operations, their MS-FP8 gives more than 90TFLOPS on the 14nm Stratix 10 280 FPGA running at 500MHz. The MS-FP8 and MS-FP9 are suitable for neural net acceleration in FPGA in Microsoft data centres.

The EIE increases performance by $2.9\times$ compared to DaDianNao, due to the sparsity of their algorithm while maintaining prediction accuracy at the 32-bit float, 32-bit integer and 16-bit integer arithmetic precision.

Substituting FP units with low precision, fixed point 16-bit to 8-bit computations are the focus of the work of Gupta *et al.*, [2015]. The researchers replace round-to-nearest with stochastic rounding for the large dot product accumulations and show little or no detrimental effect on classification accuracy. The low precision yields significant gains in energy efficiency and throughput when Gupta *et al.*, implement their accelerator in a FPGA. These computations yield a throughput of up

to 260GOPS/s at a power efficiency of 37GOPS/s/W, way above the 1-5GOPS/s/W of an Intel i7 CPU or NVidia GTX780 GPU.

With loop tiling, organisation of PE and buffer, and matching throughput of the PE with off-chip interface bandwidth Zhang *et al.*, [Zhang et al. 2015] demonstrate their implementation has a performance of 61.62GOPS and a performance density of $8.12^{-04} GOPS/slice$. These metrics are nearly four times the performance and twice the performance density of the nearest competitor with 17 GOPS and $4.5^{-04} GOPS/slice$ respectively.

Courbariaux *et al.*, significantly reducing the bit precision of the CNN to 2-bits and call their solution BinaryConnect [2015a]. The researchers reduce the multiplications by $2/3$ by forcing the weights used in the forward and backward propagations to be binary; however, values may not necessarily be 1 or 0. They achieve comparable classification results on invariant MNIST, CIFAR-10 and SVHN datasets. The bits reduction potentially reduces memory requirements by $16\times$. Courbariaux *et al.*, later coin the phrase BNN [Courbariaux et al. 2016]. BNNs can drastically reduce memory size and accesses, and reduce computation by replacing arithmetic operations with bitwise operations which substantially improves power-efficiency while having comparable classification results on invariant MNIST, CIFAR-10 and SVHN datasets. Courbariaux *et al.*, offer a GPU library of BinaryConnect online[1].

XNOR-Net [Rastegari et al. 2016] create a network of approximate weights with binary weight and input values. The estimated network gains a $32\times$ memory saving compared to a standard convolution of XNOR-Net accelerated AlexNet. The researchers approximate the convolutions in the network with bitwise operations achieving a $58\times$ speed up in inference computations compared to a standard convolution. Rastegari *et al.*, also achieve a 44.2% classification accuracy compared to the 56.7% accuracy of a standard convolution.

2(b). OPTIMISE ARITHMETIC OPERATORS:

The underlying arithmetic operators of CNNs are multiplication and accumulation, the performance of which researchers are continuously improving. To that end, Fürer [Fürer 2007] propose a faster method of multiplication. Their work compares the performance of Schönhage-Strassen's fast fourier transform (FFT) based

[1] `https://github.com/MatthieuCourbariaux/BinaryConnect`

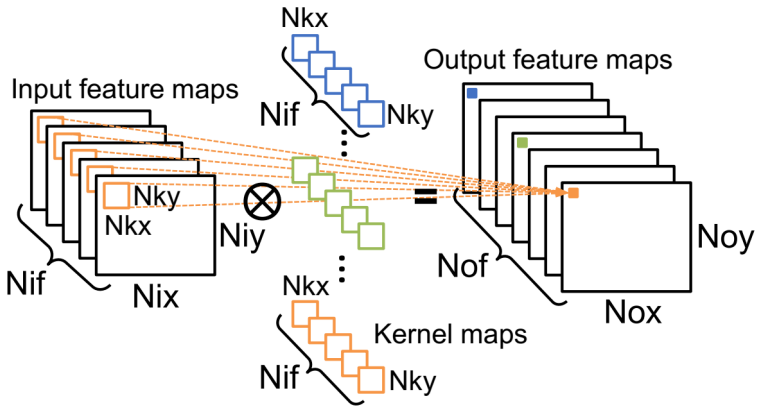


Figure 2-5: **Convolution Loop Interchange.** [Ma et al. 2017].

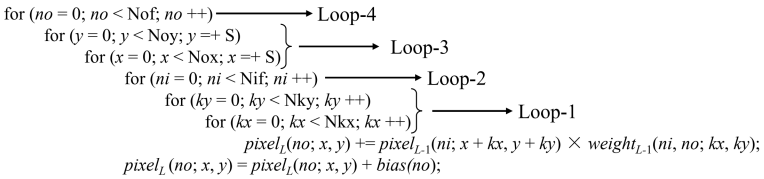


Figure 2-6: **Convolution Loop Interchange Code.** [Ma et al. 2017].

algorithm (sometimes used by CNNs), see Equation 1, which computes integers in time:

$$O(n \; log \; n \; \cdot \; log \; log \; n) \tag{1}$$

and has a lower bound of:

$$\Omega(n \; \log \; n) \tag{2}$$

to that of their algorithm that computes integers of length n in:

$$O\left(n \log n \cdot 2^{O(\log * n)}\right) \tag{3}$$

time, where $log * n$ is the iterated logarithm and thus reduces the gap between the bounds.

3. CREATE CUSTOM HARDWARE ACCELERATORS:

3(b). CREATE SPECIFIC FPGA and ASIC ACCELERATOR:

Ma *et al.*, [2017] build somewhat on the work of Zhang *et al.*, and others to propose their loop unrolling and tiling optimisation algorithm. Where Ma *et al.*, differ from Zhang [2015] is in performing loop interchange, *i.e.*, they alter the computa-

tional order of the four convolution loops, ensuring the innermost loop is the MAC within the kernel window $Nkx \times Nky$. See Figure 2-5 and the code in Figure 2-6.

2.3.5 Algorithmic Optimisation - Storage

Compression of the CNN network and weights are often a means of reducing the storage requirements of a CNN. Several researchers have focused on compression.

1. OPTIMISE THE CNN ARCHITECTURE:

1(b). COMPRESSION, PRUNING, SPARSITY:

Hinton *et al.*, with their Distilling [Hinton et al. 2015], increases sparsity in the network by building on the research of Bucilǎu *et al.*, [2006] who compress an ensemble of knowledge into a single model. Hinton *et al.*, uses a different compression technique to that of Bucilǎu *et al.'s*, work. Hinton *et al.*, show that experiments in MNIST with a distilled network that has 300 or more units in two hidden layers and their *temperature* parameter set above 8, can correctly identify a written digit 3 to a classification accuracy of 98.6%.

SqueezeNet is a much more sparse network proposed by Iandola *et al.*, [2016]. SqueezeNet is easily implementable on mobile embedded devices as SqueezeNet compresses the model parameters by $50\times$ (thus the name). Iandola *et al.*, optimise the algorithm by adopting the following three design strategies:

- Replace 3×3 filters with 1×1 filters (similar to GoogleNet and MobileNets);
- Decrease the number of input channels to 3×3 filters which they do with *squeeze layers* (a layer with only 1×1 filters);
- Downsample late in the network so that the convolution layers have large activation maps.

2. OPTIMISE THE NUMBER PRECISION AND OPERATORS OF THE CNN:

2(a). REDUCE PRECISION OF NUMBER FORMAT AND ARITHMETIC:

Researchers often quantise weight data from FP32 to FP16 and 8-bit integer and down to 1-bit binary values, thus reducing storage requirements.

Courbariaux *et al.'s*, BNN [2015a] and [2016], reduces their memory requirement by a factor of $16\times$.

Other researchers posit that using uniform bit widths of data across the entire network may be hampering efficiency. Lin *et al.*, [2016] suggest a fixed-point quan-

tisation alternative where they identify optimal fixed point bit-widths allocation across the layers of the CNN, yielding greater than 20% reduction in model size, with no loss in classification accuracy on the CIFAR-10 benchmark suite. Optimal bit-width quantisation aids the implementation of the CNN in embedded systems.

ALGORITHMIC OPTIMISATIONS CONCLUSION:

The different proposals for optimising the network and weights for storage, be it weight sharing or bit-precision alterations, allow users to select the correct network that will correspond to the memory requirements/limitations of their embedded mobile devices. The next section will look at custom bit-precision optimisations in a little more detail.

2.4 Customising Bit-Precision

The following subsections highlight changes in bit precision or number representation (see column 3 of Table 2.1), and how these changes affect classification accuracy, area, energy, execution time and storage. We will highlight various sub-categories with capitalised sub-headings.

2.4.1 Bit-Precision Optimisation - Accuracy

Researchers change bit precision to satisfy low compute or storage requirements. At the same time, researchers have to strive to maintain or in some cases, improve the top-5 and top-1 classification accuracy.

2. OPTIMISE THE NUMBER PRECISION AND OPERATORS OF THE CNN:

2(a). REDUCE PRECISION OF NUMBER FORMAT AND ARITHMETIC:

FINN discussed previously, have reduced their weights and arithmetic to a binary representation. They managed to obtain a high level of classification accuracy of CIFAR-10 and SVHN datasets of 80.1% and 94.9% respectively.

De Dinechin *et al.*, [2009] demonstrate that as the bit width of the exponent and mantissa of a FP operator can be changed to any arbitrary value, the classification accuracy of the number representation can be set to any desired accuracy.

Johnson *et al.'s*, log-linear multiply-add type that they term ELMA [2018] reaches just shy (-0.9%) of the top-1 classification accuracy of the FP equivalent ResNet-50 ImageNet validation set, and within only -0.2% of the top-5 classification accuracy.

The bit-width reduction is useful where the size of the target system for the CNN is of different size; the bit precision can be scaled based on storage or computation of the mobile device.

2.4.2 Bit-Precision Optimisation - Area

One of the advantages of using FLOating-POint COres (FloPoCo) to automate the design of customer precision arithmetic [2009], is the precise controllability it offers the designer of the area the FP arithmetic design consumes. De Dinechin *et al.'s*, results show the impact of different bit precisions of FP representation on the performance and area, when the generated RTL is synthesised with an ASIC or FPGA synthesiser. We rely on this for our HOBFLOPS proposal.

If researchers approach optimising CNNs from the bottom up and rethink how numbers are represented, then, as we discussed earlier, Johnson [2018] demonstrates a saving of between $5\times$ and $10\times$ the area of the implementation. However, this leads to upfront design costs that only larger companies may afford.

2.4.3 Bit-Precision Optimisation - Energy

1. OPTIMISE THE CNN ARCHITECTURE:

1(b). COMPRESSION, PRUNING, SPARSITY:

Energy is conserved when, either there are fewer transactions on the whole on the memory bus, or the transactions contain compressed data.

The EIE accelerator discussed above [Han et al. 2016a; Han et al. 2016b] shows their quantised multiplier dissipates around 3.75pJ and maintains a classification accuracy of around 80% of the FP32, 32-bit integer and 16-bit integer representations. The researchers push the quantisation strategy down to 4-bits. They demonstrate a power-efficient matrix-vector multiplication of 160.3GOP/s/W and claim more than an order of magnitude greater than their next nearest competitor, the DaDianNao and $24,000\times$ more efficient than CPU and $3400\times$ more efficient than GPU.

2. OPTIMISE THE NUMBER PRECISION AND OPERATORS OF THE CNN:

2(a). REDUCE PRECISION OF NUMBER FORMAT AND ARITHMETIC:

The TPU [Jouppi et al. 2017] performs at 92TOPS/s compared to the benchmark Haswell E5-2699 v3 CPU partially due to their use of bfloat16 [Google 2019].

Johnson's [2018] reduced bit precision ELMA compares to that of float16 and shows a $5\times$ power saving.

2.4.4 Bit-Precision Optimisation - Execution Time

Often reduction in bit precision is at odds with execution time. However, researchers find this not necessarily the case, if careful consideration of the underlying hardware architecture or algorithm is optimised.

1. OPTIMISE THE CNN ARCHITECTURE:

1(b). COMPRESSION, PRUNING, SPARSITY:

The reduction in bit precision of the EIE [Han et al. 2016a; Han et al. 2016b] in both the convolution and fully connected (FC) layers shows no appreciable effect on the top-1 or top-5 performance down to 16-bit integer arithmetic. They demonstrate that, compared to CPU and GPU, the EIE is $189\times$ and $13\times$ faster.

1(c). CREATE NEW LAYER TYPE WITH FEW WEIGHTS:

Howard *et al.*, [2017] compare their depthwise separable convolution to a standard baseline convolution. Their baseline gives a classification accuracy of 86.9%, which contains 1600 million multi-adds and 7.5 million parameters. Their largest equivalent 1.0 MobileNet-224 exhibits an accuracy increase of 88.7%, 568 million multiply-adds and 3.2 million parameters. Howard *et al.*, keep reducing the width of the multiplier. At 0.25 MobileNet-128, they see a drop in accuracy to 0.5% below that of the baseline. With a few arithmetic operations and parameters, the network can be implemented in a mobile device while keeping the frame rate high.

2. OPTIMISE THE NUMBER PRECISION AND OPERATORS OF THE CNN:

2(a). REDUCE PRECISION OF NUMBER FORMAT AND ARITHMETIC:

Interestingly, as discussed above, De Dinechin *et al.*, [2009] demonstrate that their 10 cycles, 368MHz slow custom FP and their 13 cycles, 368MHz fast custom FP performs faster than the equivalent 10 cycle, 319MHz 32-bit FP version, and only requires two more cycles at the same frequency of operation for their fast version.

Gupta *et al.*, [Gupta et al. 2015] recognise Courbariaux *et al.*, are doing orthogonal research [Courbariaux et al. 2015b]. Courbariaux's work proposes a hybrid of the fixed-point and the conventional floating-point arithmetic for training deep neural networks whereas Gupta's proposal trains using low-precision fixed-point arithmetic and stochastic rounding followed by fine-tuning with higher precision

fixed-point computations. When Gupta *et al.*, implement their 28×28 systolic array in a Xilinx Kintex K325T FPGA they achieve a 137GOPS/s/W performance.

INT8 [Fu et al. 2016] (mentioned above) use of 8-bit integers accommodates the bandwidth and storage of the FPGAs BRAM and DSP units. They claim the 8-bit representation gives them a competitive advantage of a peak of $1.75 : 1$ DSP multipliers to INT8 MACs ratio. Their DSP units can perform two INT8 MACs, sharing the same weights, if the multiplier is fed 24-bits and their carry accumulator with 32-bits.

Lin *et al.*, [2016] agree with Han *et al.*, that quantisation is the approach and so propose a method of converting the FP model to a fixed point quantised equivalent. The researchers propose an algorithm for the conversion but also put forward the suggestion that different layers require different bit widths for each of the AlexNet and CIFAR-10 CNNs.

The move to using TPUs [Jouppi et al. 2017] by Google for the training phase significantly improves the ML memory bandwidth of 600GB/s and performance of 45TFLOPS.

Researchers propose many approaches of customised arithmetic to reduce the arithmetic precision of CNN inference in FPGA. Some researchers suggest software-only microprocessor methods of reducing inference precision. Low-overhead software solutions can be provided if SIMD processors are targetted. Anderson *et al.*, [2017] exploit the SIMD nature of the processor with *flytes*, their scheme for byte-level customizable precision FP storage, packing and computing the SIMD vector registers. They use SIMD instructions to convert between the custom format and native FP while packing and unpacking the data into the vector registers.

Microsoft's network processing unit (NPU) [Chung et al. 2018] as part of Project Brainwave discussed above compare the performance of their MS-FP8 and MS-FP9 against 16- and 8-bit integer operations and Microsoft claim 90TOPS for MS-FP8 and 65TOPS for MS-FP9.

To decrease the training time of neural networks, one might consider using multiple processors, GPUs or FPGAs to accelerate the training. Dettmers [Dettmers 2015] suggests that the communications bottleneck between the multiple accelerators is throttling the potential speedups parallelism offers. Dettmers develop 8-bit

approximate algorithms to compress the 32-bit gradients and non-linear activations to utilise the available bandwidth and show a $50\times$ speedup compared with 32-bit on a 96 GPU accelerator system.

2(c). BITSLICING:

Xu *et al.*, [2017] propose an approach to vector computing based on bitslice vector formats and build arithmetic operators from bitwise instructions on general-purpose processors with SIMD extensions. The bitwise arithmetic approach enabled them to support a vector processing model that allowed for arbitrary precision data. Xu *et al.*, create vectors of integer or FP types of between five and eleven bits, for example, they construct a vector of thirty-two 8-bit elements, or construct a vector of thirty-two 17-bit elements. The bitwise arithmetic approach allowed optimisations of data precision on general-purpose processors that would only previously be available on FPGA or custom ASICs. Instead of storing the vectors in the traditional sense of storing, say, 17-bit vectors inefficiently in a 32-bit register, they instead store thirty-two 17-bit words that are conceptually rotated and flipped into a bitsliced format and stored in memory. The researchers use bitwise arithmetic to perform integer or floating-point arithmetic, such as addition and multiplication, on the vectors while the vectors remained in bitsliced format.

Xu *et al.*, show performances of 8- to 28-bit integer and floating-point arithmetic on the AVX2 processor from Intel. The researchers also show some real-world applications by demonstrating their bitslice arithmetic implementation in xSCALE, xAXPY, xGEMV and one-dimensional blur functions supplied in the BLAS library.

Xu *et al.*, show that for low numbers of bits up to around 11-bits on AVX2 128-bit and 14 bits for 256-bit, the massively parallel computation that bitslice arithmetic offers to deliver significant speedup of code written for bitsliced format. They do not show performance on Intel's AVX512 or any Arm device. Furthermore, their manual optimisations might be further optimised with other optimisation tools. Both of these points are explored in our HOBFLOPS work on chapter 4 on page 71.

2.4.5 Bit-Precision Optimisation - Storage

Reducing the bit precision of the pretrained weights allows for reduced storage and bandwidth costs.

1(b). COMPRESSION, PRUNING, SPARSITY:

In Deep Compression [Han et al. 2016a] that we have discussed at length above, the proposed 5.3-bits for the weights reduces to 4.1-bits when adding Huffman coding and gives a compression rate of $39\times$ for LeNet. Compare that to the other VGG-16, which uses around 6.4-bits for the weights, reduced to 4.1-bits when additionally using Huffman coding and yields a $49\times$ compression rate. AlexNet yields $35\times$, with 5.4-bits of weights, which is reduced to 4-bits with the additional Huffman coding.

Comparing Deep Compression to XNOR-Net [2016] (briefly discussed above), XNOR-Net with their binary values estimate their network gains a $32\times$ memory saving compared to a standard convolution of XNOR-Net accelerated AlexNet. However, XNOR-Net only supports CPU.

SqueezeNet [2016] performs comparisons with Deep Compression and shows for 6-bit that they reduce the storage requirements by a factor of 10 and reduce the model size vs AlexNet by $510\times$ while maintaining the same top-1 and top-5 accuracy as Deep Compression.

2.5 Conclusion

Highlighted above are three main areas of optimisations; architectural optimisations, algorithmic optimisations and bit-level optimisations. Researchers attempt to improve the level of CNN classification accuracy while improving the orthogonal areas.

Quantisation and compression techniques give excellent results and suggest further areas of improvement. We investigate exploiting and optimising weight sharing implemented in FPGA and ASIC in our PASM work, see chapter 3 on page 41.

Bitsliced arithmetic offers very good performance for low bit-widths. We investigate applying bitslicing techniques to the MAC of a CNN in high- and low-end processors in our HOBFLOPS work, see chapter 4 on page 71.

Often the effect of reducing the bit-precision of the arithmetic or increasing quantisation of the weight values is to decrease the area and energy consumed by the CPU, FPGA or ASIC and at the expense of classification accuracy. However, our works show that accuracy is not compromised with our optimisation strategies while increasing efficiency and execution time.

A further background of how CNNs work algorithmically can be found in Appendix B. There we introduce CNNs and explain how they work and how they are trained. We show the associated arithmetic to demonstrate the scope of computation needed and thus the potential optimisation avenues to explore. We find that optimisation is a big area of research, with a research priority of increasing classification accuracy and execution time of CNNs, while also attempting to optimise the models and associated data for implementation in low-power embedded IoT devices.

2.6 Central Tenets of this Research

The underlying floating- or fixed-point arithmetic of CNNs discussed in subsection B.2.4, and the number of MACs in a CNN outlined in subsection B.3.1, determines largely how much compute, energy and throughput the CNN will consume. These requirements make it challenging for researchers to implement CNNs on embedded devices. As discussed in this chapter, researchers often tackle these challenges by focusing on algorithmic and architectural optimisations to decrease execution time or area and thus energy. If CNNs are to operate on mobile edge IoT devices, then we require additional low-level optimisations, such as bit-precision optimisation to reduce energy consumption and area and increase throughput of CNNs accelerators.

Often researchers look to increase the performance of hardware CNN accelerators by implementing, in parallel, as much of the underlying gate-level logic and arithmetic as possible. The parallel hardware implementation leads to the replication of gate-level logic and arithmetic functions such as MACs. The MAC contains a multiplier which is typically large and consumes a significant quantity of the energy budget of the CNN accelerator.

Our first research area, chapter 3 on page 41, looks at reducing the number of hardware parallel multipliers in a CNN. When implemented in an ASIC, multipliers are large in gate-level area and power-hungry. Our research exploits *"weight sharing"* which allows us to replace hardware multipliers in the MAC circuit with adders and selection logic, followed by a single post-pass multiply. We implemented PASM in ASIC and analyse PASMs effectiveness. We show our approach results in fewer gates, smaller logic, and reduced power with only a slight increase in latency. We

also show PASM implemented in resource-constrained FPGAs with improved utilisation of the FPGA fabric resources.

Another method researchers employ to increase software performance of accelerators while retaining classification accuracy is to change the bit-precision of the underlying software arithmetic. Reducing bit-precision typically reduces the volume of weight data to be transferred from random access memory (RAM) and computed in CPU and GPU by the model, thus reducing the storage and energy consumption and increasing throughput of the accelerator. Hardware developers produce custom mixed precision accelerators in FPGA and ASIC but in general CPUs do not support fast, custom precision FP in software.

For our second research area, we propose HOBFLOPS, a generator of efficient custom-precision emulated bitslice-parallel software (`C/C++`) FP arithmetic, see chapter 4 beginning on page 71. We generate custom-precision FP routines, optimised using a hardware synthesis design flow, to create circuits. We provide standard cell libraries matching the bitwise operations on the target microprocessor architecture and a code-generator to translate the hardware circuits to bitslice software equivalents. We exploit bitslice parallelism to create a novel, very wide (32—512 element) vectorized CNN convolution for inference. On Arm and Intel processors, the MAC performance in CNN convolution of HOBFLOPS, Flexfloat, and Berkeley's SoftFP are compared. HOBFLOPS outperforms Flexfloat by up to $10\times$ on Intel AVX512. HOBFLOPS offers arbitrary-precision FP with custom range and precision, *e.g.*, HOBFLOPS9, which outperforms Flexfloat 9-bit on Arm Neon by $7\times$. HOBFLOPS allows researchers to prototype different levels of custom FP precision in the arithmetic of software CNN accelerators. Furthermore, HOBFLOPS fast custom-precision FP CNNs may be valuable in cases where memory bandwidth is limited.

3 Low Complexity MAC Units for CNNs with Weight Sharing

3.1 Introduction

As discussed in section A.1, page 103, CNNs are used on a daily basis for image, speech and text recognition and their use and application to different tasks is increasing at a very rapid rate. However, as discussed in section 2.1 beginning on page 9, CNNs require huge memory storage and bandwidth for weight data and large amounts of computation that would push to extremes the battery, computation, and memory in mobile embedded systems. Under the optimisation techniques of Table 2.1, researchers have proposed methods of quantising and dictionary compressing the weight data to reduce the memory bottleneck and bus bandwidth. Others have proposed various different CNN hardware accelerators implemented in both FPGAs and ASICs that may contain hundreds to thousands of parallel hardware MAC units to increase the computational performance. This increase in computational performance comes at the great expense of power as the MAC units contain a multiplier, each of which consumes large numbers of logic gates and high-power consumption in an ASIC [Sabeetha et al. 2015].

Once trained, CNNs are extensively used in an inference mode (see section A.3 starting on page 119). As discussed in section 2.1, Han *et al.*, [2016a; 2016b] found that they can prune and diction compressed similar weight values of the trained CNN. They found that tens to hundreds of weight values were sufficient while maintaining classification accuracy. Weight-sharing does not reduce the number of

MAC operations required; it reduces only the weight data storage and bandwidth requirement.

Building on the research of Han *et al.*, [2016a; 2016b], we propose a rearchitected MAC circuit of the weight-shared CNN aimed at hardware accelerators. Rather than computing the sum-of-products (SOP) in the MAC directly, we instead count how many times each of the weight indexes appears and store the corresponding IFM value in a register bin, thus replacing the hardware multipliers with counting, selection, and accumulation logic. After this weighted histogram accumulation phase, a post pass multiplication is performed of the accumulated IFM values in bins with the corresponding weight value of that bin. We call this accelerator optimisation the PASM. To evaluate PASM performance we implement PASM in a convolution layer of a weight-shared CNN accelerator. Where weight bin numbers are small, and channel numbers are large, the counting and selection logic can be significantly smaller and lower power than the corresponding multiply-accumulate circuit. We also show that PASM is beneficial when implemented in a resource-constrained FPGA as PASM consumes fewer BRAMs and DSP units for the MAC operations in the FPGA.

The rest of this chapter is organised as follows. Section 3.2 gives some background on CNN accelerators and introduces the PASM and how it compares to other CNN accelerators. Section 3.3 shows how our PASM is implemented in a convolution layer accelerator with examples compared to a weight-shared accelerator. Section 3.4 describes how a weight-shared-with-PASM convolution accelerator is designed and implemented in an ASIC at 45nm clocked at 1GHz and in a Zynq FPGA clocked at 200MHz. Section 3.5 presents the experimental results showing latency, power, and area projections for both FPGA and ASIC. Section 3.6 draws conclusions.

3.2 DNN Convolution with Dictionary-Encoded Weights

3.2.1 CNN Accelerators

As outlined in section B.3 on page 144, a CNN contains convolution layers, activation function layers such as a sigmoid or rectified linear unit (ReLU) and pooling

```
ifm[C][IH][IW], weight[M][C][KY][KX];
outFeat[M][OH][OW];

for (ihIdx=(KY/2); ihIdx<(IH-(KY/2)); ihIdx+=Stride) {
  for (iwIdx=(KX/2); iwIdx<(IW-(KX/2)); iwIdx+=Stride) {
    for (mIdx=0; mIdx<M; mIdx++) {
      summands = 0;
      for (cIdx=0; cIdx<C; cIdx++) {
        for (kyIdx=0; kyIdx<KY; kyIdx++) {
          for (kxIdx=0; kxIdx<KX; kxIdx++) {
            imVal = ifm[cIdx][((ihIdx+kyIdx)
                               -(KY/2))][((iwIdx+kxIdx)-(KX/2))];
            kernVal = kernel[mIdx][cIdx][kyIdx][kxIdx];
            summands += imVal * kernVal;
      } } }
      outFeat[mIdx][ihIdx/Stride][iwIdx/Stride] = summands;
} } }
```

Listing 3.1: **Simplified pseudo-code of a convolution layer.**

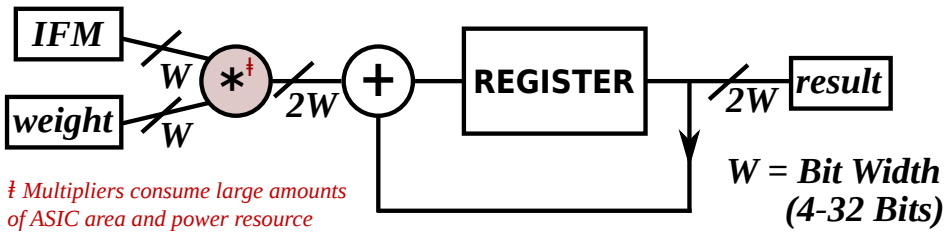


Figure 3-1: **Simple MAC block diagram.**

layers. Up to 90% of the computation time of a CNN is taken up by the convolution layers [Farabet et al. 2010]. Within the convolution layer, there are many thousands of MAC operations, as shown in the pseudo code in Listing 3.1. The convolution operator has an IFM of dimensions $IH \times IW$ and C channels and is convolved with M kernels (typically 3 to 832 [Szegedy et al. 2015]) of dimension $KY \times KX$ and C channels at a stride of S to create an output feature map of $OH \times OW$ and M channels. The loops can be unrolled into parallel MAC units and implemented in hardware [Zhang et al. 2015] to accelerate the convolution.

A MAC unit (see Figure 3-1) is a sequential circuit that accepts a pair of numeric values (IFM and weight values) of a predefined bit width and type (e.g., 32-bit fixed-point integers), computes their product and accumulates the result in the local accumulator register each clock cycle. The locality of the accumulator register reduces routing complexity and clock delays within the MAC.

In our review of the literature in section 2.1 on 9, Han *et al.*, [2016a; 2016b] propose a weight-sharing architecture to reduce the power and memory bandwidth consumption of CNNs. They found that similar weight values occur multiple times in a trained CNN. By binning the weights and retraining the network with the

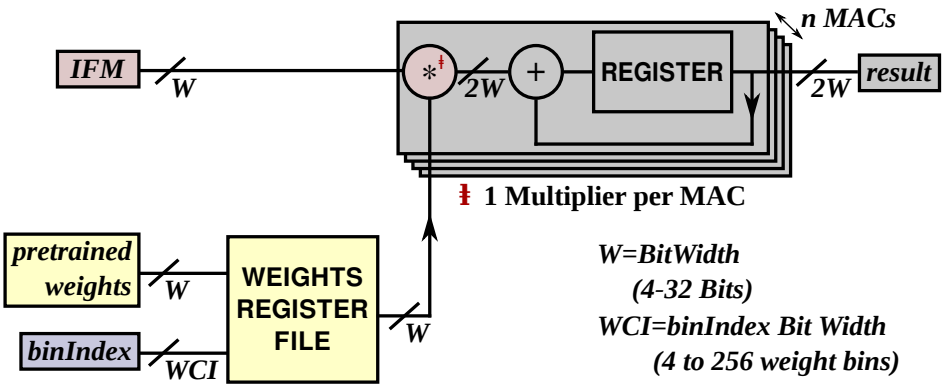


Figure 3-2: **Simplified weight-shared MAC block diagram.**

binned values, they found that just sixteen weights were sufficient in many cases. They encode the weights by replacing the original numeric values with a 4-bit index that specifies which of the sixteen shared weights should be used. This greatly reduces the size of the weight matrices. Figure 3-2 shows simplified weight-sharing decode logic coupled with multiple MAC units of the CNN. When the kernel input is encoded using weight-sharing, an extra level of indirection is required to index and access the actual weight value from the weights register file.

Figure 3-3 shows an example of the weight-shared MAC in operation. Each ***IFM*** value is streamed in, and its corresponding ***binIndex*** is used to access the pretrained weight against which to multiply and accumulate into the result register. Figure 3-3 shows how ***IFM*** value 26.7 is multiplied-accumulated with the pretrained weight 1.7 indexed by ***binIndex*** 0. Next 3.4 is multiplied-accumulated with the pretrained weight 0.4 indexed by ***binIndex*** 1. This continues until finally multiplying-accumulating IFM value 6.1 with pretrained weight value 1.7 of bin 0 to give the result of Equation 1.

$$\textbf{\textit{result}} = (26.7 \times 1.7) + (3.4 \times 0.4) + (4.8 \times 1.3) + (17.7 \times 2.0) + (6.1 \times 1.7) = 98.8. \tag{1}$$

In both a simple MAC (Figure 3-1) and a weight-shared MAC (Figure 3-2) the multiplier is the most expensive unit in terms of floor area (i.e. large numbers of gates) and power consumption in an ASIC or numbers of DSP units in an FPGA. As a large number of MAC units are used in a parallel weight-shared CNN hardware accelerator, the overall area and power is likely to be large.

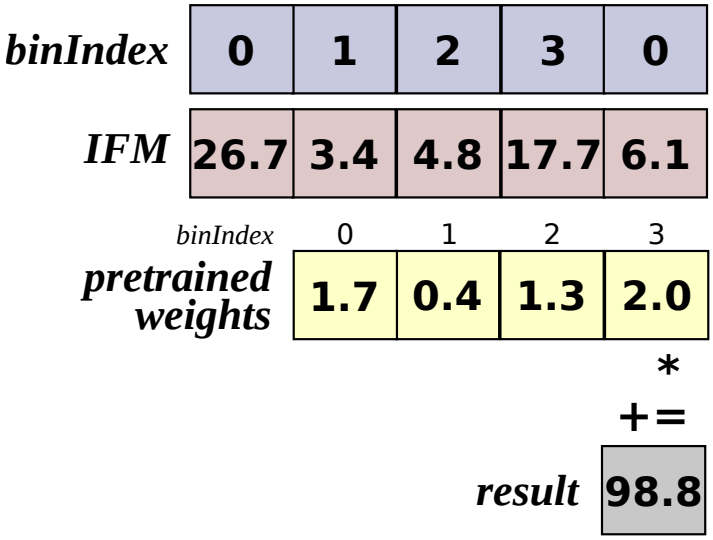


Figure 3-3: **Simplified weight-shared MAC example.**

Weight-sharing is an important factor in implementing CNN accelerators in an off-line embedded, low power device (see section 2.1). Han *et al.*, [2016a; 2016b] show that when pruning, quantisation, weight-sharing and Huffman coding are all used together in an AlexNet [Krizhevsky et al. 2012] CNN accelerator, the weight data required are reduced from 240MB to 6.9MB, a compression factor of $35\times$. Unfortunately, they do not provide results for the effect of weight-sharing alone, without these other optimisations. When they apply similar pruning, quantisation, weight-sharing and Huffman coding to the VGG-16 [Simonyan and Zisserman 2014b] CNN accelerator, the weight data is reduced from 552MB to 11.3MB, a $49\times$ compression ratio. The fully connected layers dominate the model size by 90%, but Han *et al.*, [2016a; 2016b] show that these layers compress by up to 96% of weights pruned in VGG-16 CNN. These newly weight-shared CNNs run $3\times$ to $4\times$ faster on a mobile GPU while using $3\times$ to $7\times$ less energy with no loss in classification accuracy. As the number of free parameters being learned is reduced in a weight-shared CNN, the learning efficiency is greatly increased and allows for better generalisation of CNNs for vision classification.

The trend is towards increasingly large networks, increasing the number of layers such as ResNet [He et al. 2015] or increasing the convolution types within each layer such as GoogLeNet [Szegedy et al. 2015] (see section B.3 for more details). Weight-sharing is one method that is getting increased research focus to reduce the

overall weight data storage and transfer so that the networks can be implemented on off-line mobile devices.

CNN hardware accelerators typically use 8-, 16-, 24- or 32-bit fixed-point arithmetic [Chen et al. 2016]. A combinatorial W-bit multiplier requires $O(W^2)$ logic gates to implement which makes up a large part of the MAC unit. Note that sub-quadratic multipliers are possible but are inefficient for practical values of W [Fürer 2007].

3.2.2 The PASM Concept

We propose to reduce the area and power consumption of MAC units by rearchitecting the MAC to do the accumulation first, followed by a shared post-pass multiplication. Our new PASM accelerator is shown in Figure 3-4. Rather than computing the SOP in the MAC directly, PASM instead counts how many times each B-bin weight-shared index appears and accumulates the corresponding W-bit ***IFM*** value in the corresponding B weight-shared bin register indexed by the ***binIndex***. PASM has two phases:

1. accumulate the IFM values into the weight bins (known as the parallel accumulate and store (PAS));

2. multiply the binned values with the weights (completing the PASM).

Figure 3-5a shows an example of the accumulation phase. Our PAS unit is a sequential circuit that consumes a pair of inputs each cycle. One input is an ***IFM*** value, and the other is the ***binIndex*** of the weight value in the dictionary of weight encodings. The PAS unit has a set of B accumulators, one for each entry in the dictionary of weight encodings. The accumulators are initially set to zero. Each time the PAS consumes an input pair, it adds the ***IFM*** value to the accumulator with index ***binIndex***. For example, when the leftmost pair of inputs in Figure 3-5a are consumed, the ***IFM*** value 26.7 is added onto accumulator numbered ***binIndex*** $= 0$. Next 3.4 is accumulated into bin 1. This continues until finally accumulating 6.1 into bin 0 to give $26.7 + 6.1 = 32.8$. This accumulated result tells us that the weight stored in dictionary location 0 has been paired with an accumulated ***IFM*** value of 32.8. For the accumulation phase, the actual weight value stored in dictionary location 0 does

not matter. We are simply computing a weighted histogram of the dictionary weight indices.

In the second phase, the histogram of weight indices is combined with the actual weight values to compute the result of the sequence of multiply-accumulate operations. Figure 3-5b demonstrates the multiply phase, multiplying-accumulating bin 0 ***pretrained weight*** with bin 0 accumulated ***IFM*** value, giving $32.8 \times 1.7 = 55.76$. The contents of ***pretrained weight*** bin 1 is multiplied-accumulated with ***IFM*** bin 1 value and so on until all the corresponding bins are multiplied-accumulated into the ***result*** register, giving 98.8, the same result found by the weight-shared MAC, Figure 3-3.

This second multiply stage can be implemented using a traditional MAC unit that is shared between several PAS units. Several MAC units can be replaced by the same number of PAS units sharing a single MAC. For example, consider the case where we must compute many multiply-accumulate sequences, where each sequence consumes $1,024$ pairs (IFM and weight) of values. A fully-pipelined MAC unit is a sequential circuit that will typically require a little over $1,024$ cycles to compute the result.

If we have four such MAC units, we can compute four such results in parallel, again in around $1,024$ cycles. If the weight data has been quantised and dictionary encoded to, say, sixteen values, then we could use PAS units with $B = 16$-bins to perform the accumulate phase of the PASM computation. Four such fully-pipelined PAS units could perform the accumulation phase in around $1,024$ cycles. However, the accumulation phase of the PASM does not give us a complete answer. We also need to perform the multiply phase, which involves multiplying and accumulating $B = 16$ values in this example. If each PAS unit had a MAC unit, then the multiply phase would take around sixteen cycles for a total of $1,024 + 16 = 1,040$ cycles for the entire multiply-accumulate operation. However, in this example, the four parallel PAS units share a single MAC unit with the result that the total time will be $1,024 + 4 \times 16 = 1,088$ cycles. PASM can have higher throughput when compared to the standard MAC due to the PAS units being much smaller than the MAC for small values of B, up to about $B = 16$.

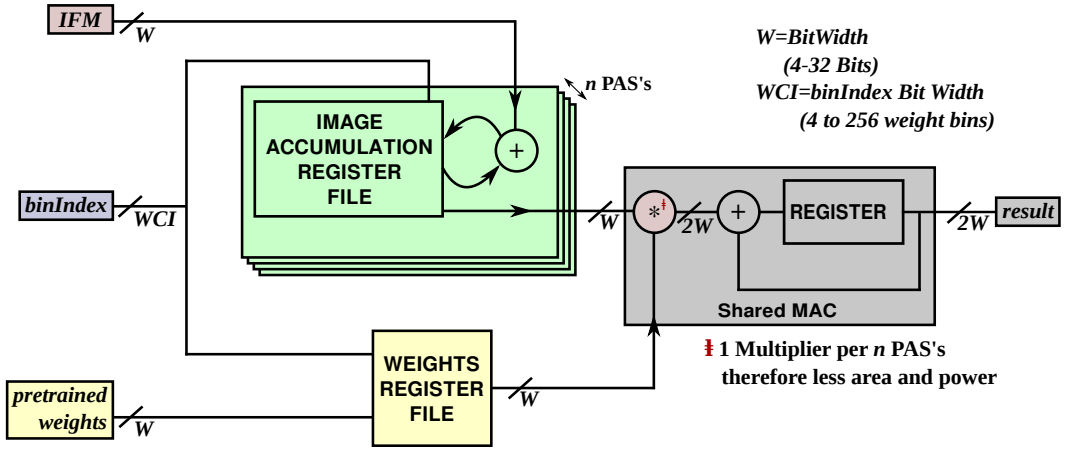
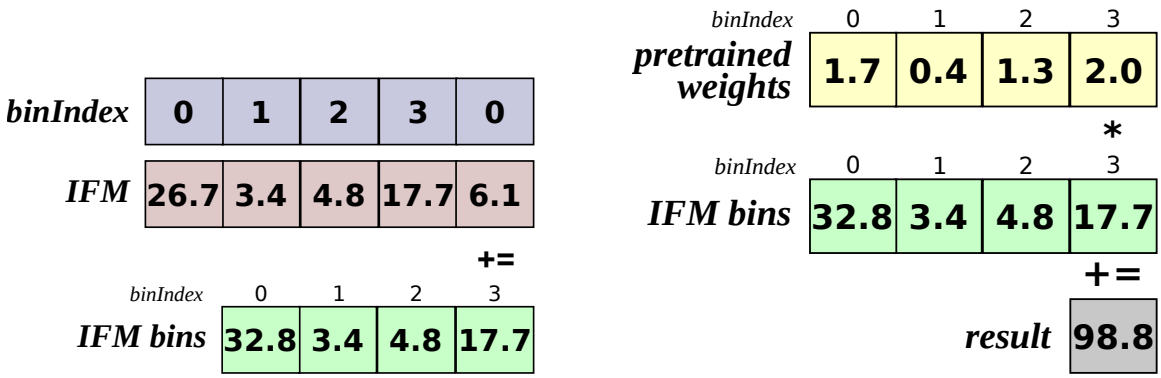


Figure 3-4: **PASM showing PAS unit followed by a shared MAC.**



(a) **Phase 1: As each IFM value is streamed in, its associated bin index is also streamed so that the IFM values can be accumulated into correct bins.**

(b) **Phase 2: Each bin accumulated value is multiplied with its corresponding pretrained weight value to produce the final result.**

Figure 3-5: PASM in Operation.

3.2.3 PASM accelerator

Table 3.1 compares the gate counts of the sub-components of a simple MAC, a weight-shared MAC and a PAS. The *gates* column shows the circuit complexity in gates of each sub-component, assuming fixed-point arithmetic. The bit-width of the data is W, and the number of bins is B in the weight-shared designs. For example, a simple MAC unit contains an adder ($O(W)$ gates), a multiplier ($O(W)^2$) gates) and a register ($O(W)$ gates). A weight-shared MAC also needs a small register file with B entries to allow fast mapping of encoded weight indices to shared weights. The PAS needs a read and write port due to the interim storage of the accumulation results

Table 3.1: Complexity of MAC, Weight-shared MAC and PAS.

Sub Component	Gates	Simple MAC	Weight-Shared MAC	PAS
Adder	$O(W)$	1	1	1
Multiplier	$O(W^2)$	1	1	
Weight Register	$O(W)$	0	B	
Accumulation Register	$O(W)$	1	1	B
File Port	$O(WB)$		1	2

that need to be read by the post pass multiplier, whereas the MAC only needs a write port.

From Table 3.1, we can also see that the efficiency of PAS depends on a weight-sharing scheme where the number of bins, B, is much less than the total number of possible values that can be represented by a weight value, that is 2^W. For example, if we consider the case of $W = 16$, then in the absence of weight-sharing, a PAS would need to deal with the possibility of 2^{16} different weight values, requiring 2^{16} separate bins. The hardware area of these bins is likely to be prohibitive. Therefore, PAS is effective where the number of bins is much lower than 2^W.

3.2.4 Evaluation of PASM as a Stand-alone Unit

We design an accelerator unit to perform a simplified version of the accumulations in Figure 3-4. Our accelerator accepts four IFM inputs and four shared-weight inputs each cycle and uses them to compute sixteen separate MAC operations each cycle. The weight-shared version performs these operations on sixteen weight-shared MAC units (16-MAC). Our proposed PASM unit has sixteen PAS units and uses four MAC units for post-pass multiplication (16-PAS-4-MAC). Both the weight-shared and weight-shared-with-PASM accelerators are coded in Verilog 2001 and synthesised to a flat netlist at 100MHz with a short 0.1ns clock transition time targeted at a 45nm process ASIC. We measure and compare the timing, power, and gate count in both designs for the same corresponding bit widths and the same numbers of weight bins.

The standard 16-MAC and the proposed 16-PAS-4-MAC each have W-bit ***IFM*** and ***weight*** inputs and the 16-PAS-4-MAC has a WCI-bit ***binIndex*** input to index into the $B = 2^{wci}$ weight bins. The designs are coded using integer/fixed point precision numbers. Both versions are synthesised to produced a gate level netlist and

timing constraints designed using Synopsys design constraint (SDC) [Gangadharan and Churiwala 2013] so that both designs meet timing at 100MHz.

Cadence Genus (version 15.20 - 15.20-p004_1) is used for synthesising the RTL into the Oklahoma State University (OSU) FreePDK 45nm process ASIC and applying the constraints in order to meet timing. Genus supplies commands for reporting approximate timing, gate count, and power consumption of the designs at the post-synthesis stage. The "report timing," "report gates," and "report power" commands of Cadence Genus are used to obtain the results for both 16-MAC and 16-PAS-4-MAC accelerators. Graphs of the gate count and power consumption results are produced for the two different designs at different bit widths and different numbers of weight bins, showing that the PASM is consistently more area and energy efficient than the weight-sharing MAC.

Figure 3-6 shows comparisons of the logic resource requirements of a $B = 16$ shared-weight-bin 16-PAS-4-MAC and 16-MAC for varying W-bit widths. Gate counts are normalised to a NAND2X1 gate. The PASM uses significantly fewer logic gates. For example, for $W = 32$-bit wide the 16-PAS-4-MAC is 35% smaller in sequential logic, 78% smaller in inverters, 61% smaller in buffers and 68% smaller in logic, an overall 66% saving in total logic gates. The PASM requires more accumulators for the B-entry register file, but otherwise overall resource requirements are significantly lower than that of the MAC.

Figure 3-7 shows comparisons of power consumption of the accelerators. 16-PAS-4-MAC's power is lower than the weight-shared 16-MACs and the gap grows with increasing W-bit width. For example, for the $W = 32$-bit versions of each design, the 16-PAS-4-MAC consumes 60% less leakage power, 70% less dynamic power and 70% less total power than that of the 16-MAC version.

Figure 3-8 shows the effect of varying the number of bins from $B = 4$ to $B = 256$, with gate counts normalised to a NAND2X1. For bit width $W = 32$ and $B = 16$-bins the 16-PAS-4-MAC utilisation has 35% fewer sequential gates, 78% fewer inverters, 62% fewer buffers and 69% fewer logic and 66% less total logic gates compared to the 16-MAC design. However, at $B = 256$, PASM registers and buffers are less area efficient than the MAC.

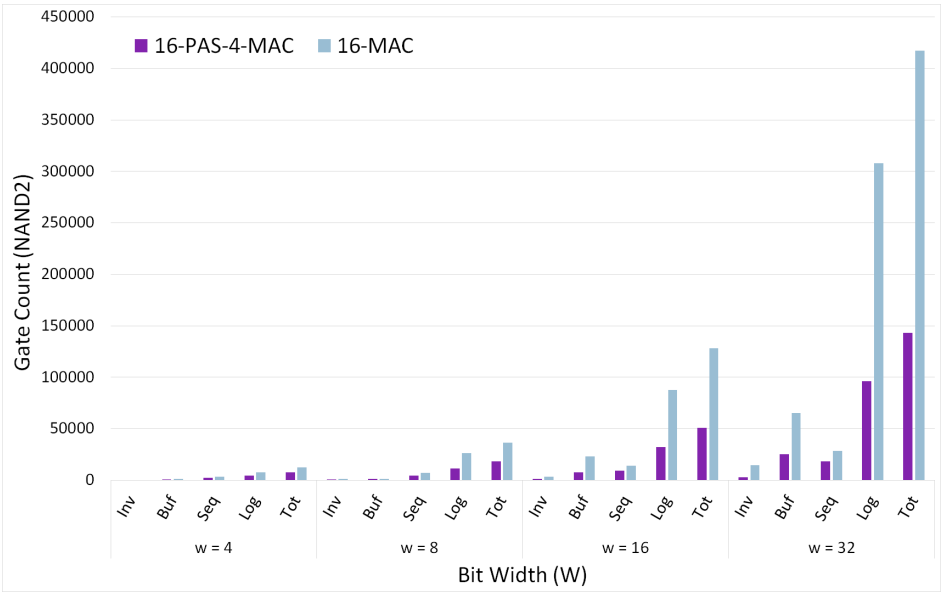


Figure 3-6: **Logic gate count comparisons (in NAND2X1 gates) for $W = 4, 8, 16, 32$-bits wide 16-MAC and 16-PAS-4-MAC for $B = 16$ weight bins; lower is better.**

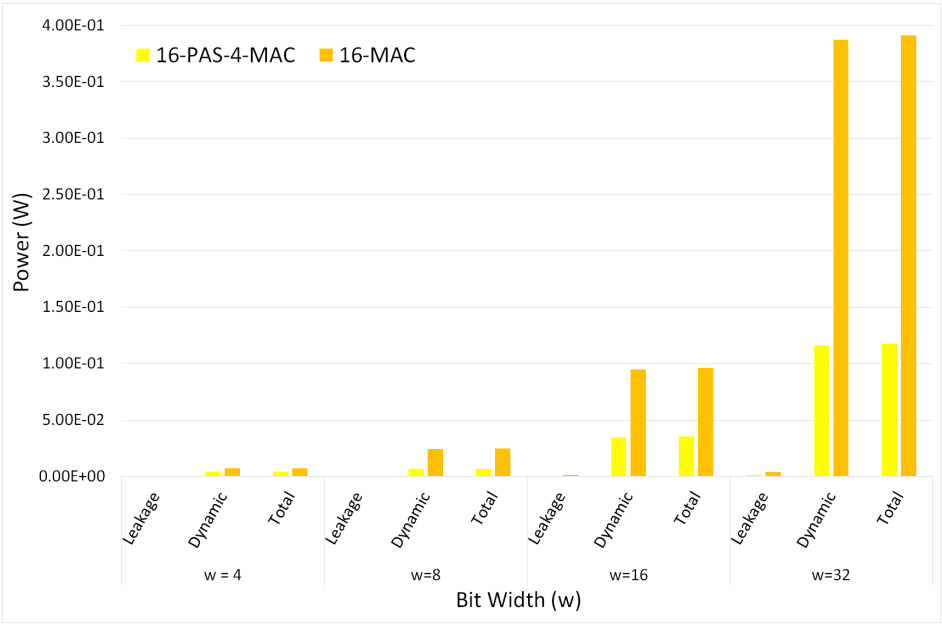


Figure 3-7: **Power consumption (in W) comparisons for $W = 4, 8, 16, 32$-bits wide 16-MAC and 16-PAS-4-MAC for $B = 16$ weight bins; lower is better.**

The 16-PAS-4-MAC also consumes 61% less leakage power, 70% less dynamic power, and 70% less total power (Figure 3-9). More details can be found in our original article, [Garland and Gregg 2017].

3.3 PASM in a CNN Accelerator

In this chapter, we asked the question would PASM offer similar power and area savings when implemented in a layer of a CNN accelerator and how would it affect

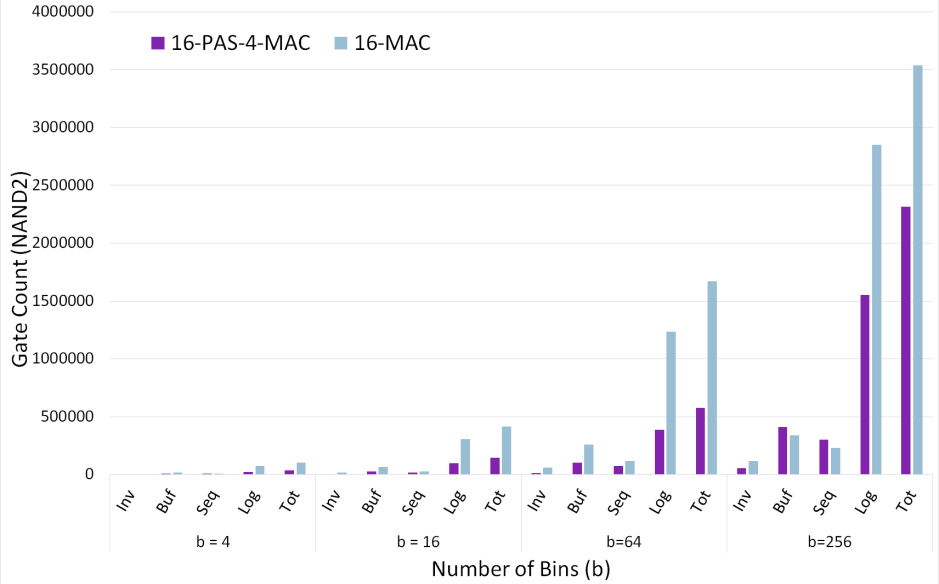


Figure 3-8: **Logic gate counts comparisons (in NAND2X1 gates) for $B = 4, 16, 64, 256$ weight bins for a 16-MAC and 16-PAS-4-MAC for $W = 32$-bit width; lower is better.**

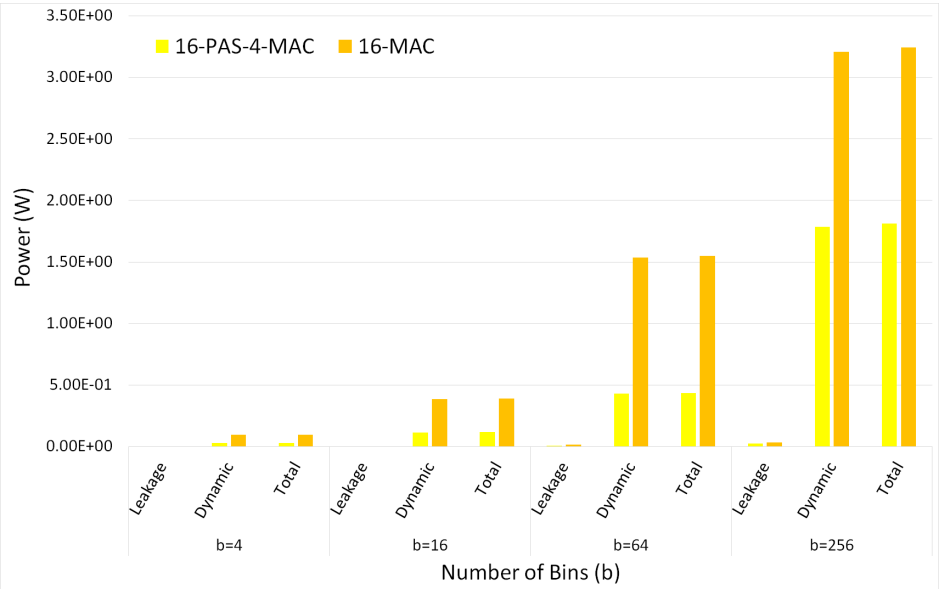


Figure 3-9: **Power consumption (in W) comparisons for $B = 4, 16, 64, 256$ weight bins deep 16-MAC and 16-PAS-4-MAC for $W = 32$-bit width; lower is better.**

performance of the convolution accelerator? We attempt to answer this by implementing PASM in a weight-shared convolution layer accelerator and evaluate and compare its latency, power, and area performance with a weight-shared convolution accelerator and baseline both against a non-weight-shared convolution accelerator for the same clock speed. Figure 3-11 shows how, when PASM is implemented in a weight-shared convolution accelerator, multiple PAS units are created in parallel

to accelerate the accumulation of $C \times IH \times IW$ ***IFM*** data into the corresponding B-bin registers. Multiplexers are created to expand and parallelise the ***IFM*** and ***binIndex*** data and demultiplexers then combine the PAS outputs for the post-pass MAC. The post-pass MAC multiplies and accumulates the binned ***IFM*** data with the corresponding $M \times C \times KX \times KY$ shared-weight value into the $M \times IH \times IW$ ***outFeat***.

The ***IFM*** data of $C \times IH \times IW$ are buffered in registers, ***weight*** data of $M \times C \times KX \times KY$ are buffered in shared weight registers, the ***binIndex*** data up to sixteen values are registered, and finally, the output feature map of $M \times IH \times IW$ is registered in an ***outFeat*** register file. This allows for greater locality and reuse of the data.

As can be seen from Table 3.1 and Table 3.2, the PASM is only efficient when the number of PAS units created is much smaller than the number of items to accumulate, i.e., the PASM is efficient only where the number of bins, B, is much smaller than the number of pairs of inputs to be multiplied and summed, Equation 2. In the absence of quantisation and weight-sharing, the PASM would not be viable. For example, if we tried to use PASM for 16-bit weight values without using quantisation or weight-sharing, then we would need 2^{16} bins in the PASM. A PASM unit with so many bins would not be competitive with a conventional MAC unit.

Any weight-shared network such as a weight-shared AlexNet [Krizhevsky et al. 2012], weight-shared VGG [Simonyan and Zisserman 2014b] or weight-shared GoogLeNet [Szegedy et al. 2015], and more generally regional CNNs, recurrent neural networks (RNNs) and long short term memorys (LSTMs) are possible good candidates for the use of PASM, although the evaluation in these networks is beyond the scope of this chapter.

3.3.1 Examples

For a simplified weight-shared accelerator, Figure 3-10, each kernel channel is "slid" across the corresponding IFM channel, multiplying and accumulating each of the pixel values with the kernel's pretrained weight-shared values into the corre-

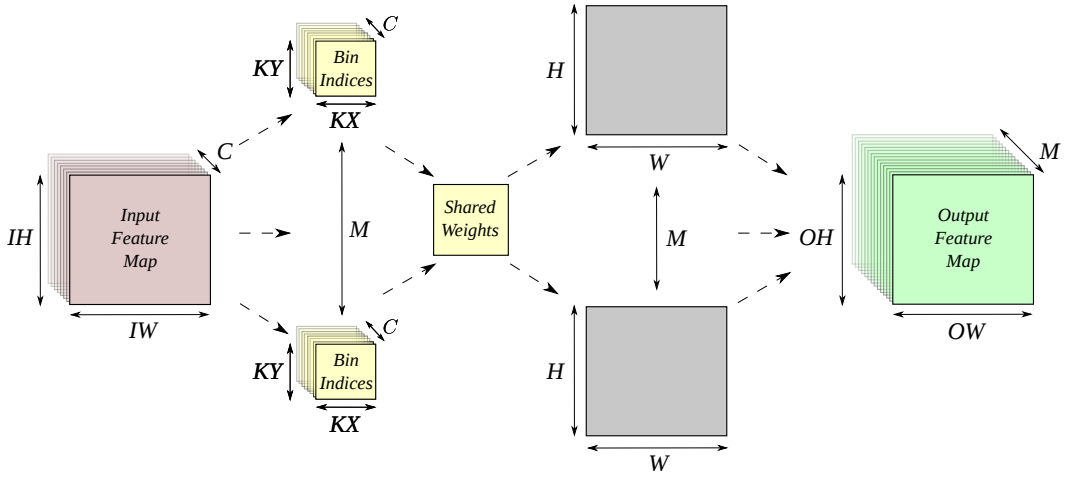


Figure 3-10: **Example of a simplified weight-shared convolution.**

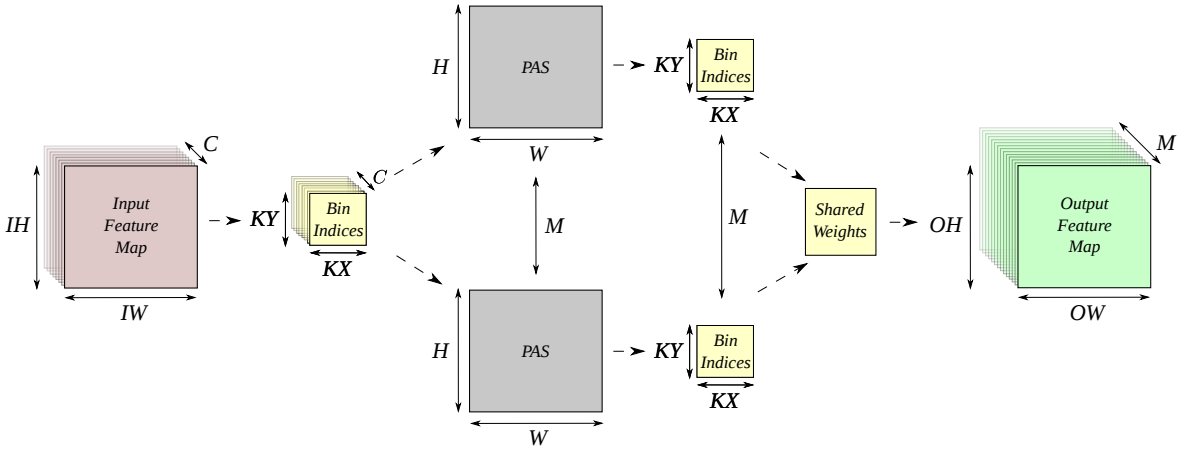


Figure 3-11: **Example of a simplified weight-shared convolution with PASM.**

sponding interim feature map channel. Each of the interim feature map channels is then "stacked" to produce the output feature map.

Now assume a simplified weight-shared-with-PASM accelerator with the same number of channels and kernels, Figure 3-11. Again, each kernel channel is "slid" across the corresponding IFM channel; however, the "kernel" contains bin indices that address the interim feature map bin into which the IFM pixel values are accumulated. After all the IFM channels have been accumulated into the IFM bins of the interim feature map, the bin indices are "slid" across the interim feature map, multiplying each of the accumulated IFM values with the corresponding kernel's in-

```
bi[C][KY][KX], sk[B], bias[M], ifm[C][IH][IW],
ifmBin[B], outFeat[M][OH][OW];
#pragma HLS ARRAY_PARTITION variable=ifmBin complete dim=1
#pragma HLS ALLOCATION instances=mul limit=1 operation
for (ihIdx=(KY/2); ihIdx<(IH-(KY/2)); ihIdx+=Stride) {
  for (iwIdx=(KX/2); iwIdx<(IW-(KX/2)); iwIdx+=Stride) {
    for (mIdx=0; mIdx<M; mIdx++) {
#pragma HLS PIPELINE II=1 rewind
      // Reset the ifmBin register file
      for (bin=0; bin<B;bin++) {
#pragma HLS UNROLL
#pragma HLS LOOP_MERGE
        ifmBin[bin]=0;
      }
      binIdx=0;
      // For each channels, stride kernel-sized-bin indices over IFM
      // accumulate IFM value in corresponding ifmBin PAS
      for (cIdx=0; cIdx<C; cIdx++) {
        for (kyIdx=0; kyIdx<KY; kyIdx++) {
          for (kxIdx=0; kxIdx<KX; kxIdx++) {
            imVal=ifm[cIdx][((ihIdx+kyIdx)-(KY/2))][((iwIdx+kxIdx)
                                                    -(KX/2))];
            binIdx=bi[cIdx][kyIdx][kxIdx];
            ifmBin[binIdx] += imVal;
      } } }
      // Once looped over channels, stride kernel-sized-bin indices
      // over PAS, multiply with corresponding shared-weight value
      cIdx=0;
      for (kyIdx=0; kyIdx<KY; kyIdx++) {
        for (kxIdx=0; kxIdx<KX; kxIdx++) {
          mul[cIdx][kyIdx][kxIdx] =ifmBin[bi[cIdx][kyIdx][kxIdx]] *
                                        sk[bi[cIdx][kyIdx][kxIdx]];
      } }
      // Sum all channel's together into each OFM channel
      for (cIdx=0; cIdx<C; cIdx++) {
        for (kyIdx=0; kyIdx<KY; kyIdx++) {
          for (kxIdx=0; kxIdx<KX; kxIdx++) {
            outFeat[mIdx][ihIdx/Stride][iwIdx/Stride] +=
                                         mul[cIdx][kyIdx][kxIdx];
} } } } } }
```

Listing 3.2: **Simplified SystemC code for the weight-shared-with-PASM convolution.**

dexed pretrained weight-shared values, and accumulated into the associated output feature map channel.

Listing 3.2 shows the simplified SystemC code for weight-shared-with-PASM implemented within a convolution layer. It demonstrates an ***IFM*** of $C \times IH \times IW$, a ***kernel*** of $M \times C \times KY \times KX$, with B ***weight*** bins, a ***stride*** of S, and an ***outFeat*** of $M \times OH \times OW$.

3.4 Design and Implementation of the PASM CNN Accelerator

For comparison, three versions of the accelerator, a non-weight-shared, a weight-shared and a weight-shared-with-PASM accelerator, are designed and synthesised. The accelerators are coded in SystemC which allows the designs to be partitioned, unrolled and pipelined to optimise power and area (NAND2 equivalent gate count) by using SystemC *#pragma* directives rather than having to hand-code the partitioning, unrolling, and pipelining in Verilog.

To increase the throughput of the PAS phase of the weight-shared-with-PASM CNN accelerator, the $IFMBin$ array of line 12 in Listing 3.2 is partitioned completely using the directive *ARRAY_PARTITION dim=1* (see line 2) to inform Xilinx Vivado_HLS to implement all bins in registers. When the *for* loop of line 9 to line 13 is unrolled using the directive *UNROLL* (see line 10) and loop merged using the directive *LOOP_MERGE* (see line 11), Vivado_HLS implements $IFMBin$ in registers rather than BRAM, allowing the high-level synthesis (HLS) to create multiple copies of the loop body so that it can parallelise the accumulation registers and associated accumulator logic and thus reduce the number of clock cycles of reads and writes to the $IFMBin$ registers.

The rest of the loops including the post pass MAC loop on lines 33 and 42 are pipelined with the directive *PIPELINE II=1 rewind* that has an iteration interval of 1, suggesting to Vivado_HLS that the loops shall need to process a new input every cycle that Vivado_HLS will try to meet if possible. The *rewind* option is used with the pipeline function to enable continuous loop pipelining such that there is no pause between one loop iteration ending and the next beginning. This is effective as there are perfect nested loops in the convolution.

The partitioning, unrolling and loop merging reduces the latency cycles of the non-weight-shared, weight-shared and weight-shared-with-PASM accelerators by 92% at the expense of increasing the flip flop count by 97% and thus the power and area of these combined function and loop pipeline registers. Implementing the $IFMBin$ array in registers allows for cell compatibility in the ASIC synthesis tool and quick synthesis time as no SRAM needs to be modelled and implemented to

store the input ***IFM*** and ***outFeat*** values. This increased power and area overhead of the accelerators is a good trade-off for the increased throughput and lower latency.

As outlined in section B.3 on page 144, the three versions of the CNN accelerators are based on the AlexNet [Krizhevsky et al. 2012] CNN and accelerate one layer of the convolution to allow for implementation in an FPGA. The accelerators include stride, an activation function, ReLU, and bias (a means for the network to learn more easily) as the activation function and bias parameters are not shared. Striding (lines 4, 5, and 42 of Listing 3.2 allows for compression of the IFM or input feature map by allowing differing pixel strides of the kernel across the input feature map. For a stride value of 1, the kernel is moved across the input feature map at a stride of one pixel at a time. With a stride of 2 or more, the kernel jumps two or more pixels as the kernel strides across the feature map. This sliding of the kernels produces smaller spatial output feature maps. The use of PASM in the weight-shared accelerator is transparent to the functionality of the stride, activation function or biasing. Note that the numbers of weight parameters for a weight-shared system must be clustered (usually with K-means) and quantised to fit into 16- to 256-bins (see Han *et al.*, [2016a; 2016b] research), as this reduction in numbers of weights is what allows the PASM reduction in power and area by doing the PAS accumulations first followed by a single post pass MAC.

Our accelerators are high-level-synthesised to a hierarchical Verilog netlist using Xilinx Vivado_HLS (version 2017.1), which allowed for quick functional simulation and hardware co-simulation and could also allow for implementation in both ASIC and later FPGA. Vivado_HLS reports the approximate latency of the design along with the approximate utilisation results for BRAM, DSP, flip flops, and look up tables (LUTs) after high-level synthesis has been executed.

When implementing the accelerators in FPGA, Xilinx Vivado (version 2017.1) is used to synthesise, optimise, place, and route the netlist from Xilinx Vivado_HLS into a Xilinx 7-series Zynq XC7Z045 FPGA part running at 200MHz. When implementing the accelerators into a 45nm process ASIC running at 1GHz, Cadence Genus is used to synthesise and optimise the design for ASIC. Cadence Genus supplies commands for reporting approximate timing, gate count and power consumption of the designs at the post-synthesis stage. The “report timing,” “report gates,”

and "report power" commands of Cadence Genus are used to obtain the ASIC timing, gate count, and power results. The gate count is normalised to a NAND2 gate, and this number reported as the overall gate count.

The designs are coded using integer/fixed-point precision numbers (INTs). The bit widths of the IFM are maintained at 32-bit INTs while the weights are stored as variable 8-bit, 16-bit, and 32-bit INTs. The bin indexes are stored as 2^2-bits for four weights up to 2^4-bits for sixteen weights.

The encoding of finite state machines is set to grey encoding to keep the power consumption of the designs to a minimum. All registers and memories in the accelerators derived from variables in the SystemC are reset or initialised to zero. The resets are set as active low synchronous resets.

The number of kernels M is kept small, i.e., $M = 2$ to keep the synthesis time the ASIC tools to a minimum. The number of channels C is made as large as possible such that the N of Equation 2 is larger than B-bins to demonstrate the power saving effect of PASM compared to the same number of channels for the weight-shared version of the accelerator, as suggested in Table 3.1 and demonstrated in Table 3.2.

The IFM cache was kept to a small tile of the IFM of multiple channels ($IH = 5, IW = 5, C = 15$) to allow its implementation in a register file. However, the IFM cache could be implemented in SRAM in an ASIC. This would allow for larger cache storage of IFM and weight values and further reduce the power and area of the accelerators but would require more "back-end" layout design work of the accelerator, something not considered for this chapter[1]. The $binIndex$ would remain in a register file as a maximum of 16×32-bit values would be stored.

To further ensure the lowest number of multipliers utilised in the PASM accelerator, the *ALLOCATION* directive is used to ensure that only one post pass multiplier is used which further reduces the area and power while very slightly increasing the latency.

[1]The OSU FreePDK 45nm ASIC process cell library used for the experiments does not have a facility to synthesise on-chip SRAM in our implementation of the weight-shared-with-PASM accelerator. If we had access to a library that would allow SRAM synthesis, then we would be able to operate on larger data blocks in our ASIC design. The weight-shared-with-PASM is likely to be even more effective with larger input blocks (particularly a large value of C) because the cost of the post-pass multiplication can be amortised over more inputs.

The Verilog netlists that are produced by Xilinx Vivado_HLS are synthesised for ASIC to produce a gate-level netlist. Timing constraints in SDC are created [Gangadharan and Churiwala 2013] so all versions of the accelerator meet timing at an iso-frequency of 1GHz with a short 0.01ns clock transition using Cadence Genus (version 17.11) synthesiser.

The synthesis targets the OSU FreePDK 45nm ASIC process cell library. Timing, latency, gate count (normalised to a NAND2 gate) and power consumption at different B-bins and W-bit widths are captured. These values are approximations as they are the post-synthesis estimates. The values will be optimised when implemented in ASIC or FPGA.

The weight-shared-with-PASM introduces a delay in processing the output of the PAS units. The PAS unit has a throughput of one pair of inputs per cycle, and so computes the initial accumulated values in about N cycles, where

$$N = (KX \times KY) \times C \tag{2}$$

The post pass MAC unit also has a throughput of one pair of inputs per cycle, so requires one cycle for each of the B accumulator bins, for a total of $N + B$ PASM cycles. In contrast, a simple MAC unit requires just N cycles, however, consumes significantly more area and power, when compared to an accelerator with more than one PAS per MAC.

Table 3.2 shows the number of MAC operations that contribute to each output for various values of C and KX and KY. For example, if $C = 32$ input channels are used with kernels of dimensions $KX \times KY = 5 \times 5$, then each computed value will be the result of 800 MAC operations. A simple fully pipelined MAC unit might be able to compute this result in a little more than 800 cycles. As can be seen from lines 11 to 13 of Listing 3.1, each element of the output of a convolution layer of a CNN is the result of the Equation 2 multiply-accumulate operations or 800 cycles in this example.

In contrast, a PASM has two phases: a PAS phase and a post-pass MAC phase. The PAS phase computes a histogram of the frequency of each weight input and depends entirely on the number of inputs. However, the post-pass MAC phase depends not on the number of inputs but on the number of different weights that

Table 3.2: Typical Numbers of MAC Operations.

		input_channels (C)		
		32	**128**	**512**
kernels (K)	**1x1**	32	128	512
	3x3	288	1152	4608
	5x5	800	3200	12800
	7x7	1568	6272	25088

can appear (each of which occupies one of the B-bins). Provided the number of inputs, N (see Equation 2), is much larger than the number of bins, B, the cost of the post-pass remains small relative to the cost of the PAS phase. For example, if $B = 16$, then the cost of the post-pass will be a small fraction of the 800 operations needed at the PAS phase. Careful consideration of the size of bins used with respect to the number of channels and kernels is important due to the $summands$ being multiplied-accumulated many times before the ***outFeat*** is updated as can be seen on lines 11-13 of Listing 3.1. The number of accumulations should therefore be much larger than B for PASM to be area and energy efficient in a weight-shared convolution accelerator.

3.5 Evaluation of PASM in a CNN accelerator

3.5.1 ASIC Results

The PASM is implemented in a weight-shared CNN accelerator and synthesised into an ASIC. The latency is compared with that of the weight-shared accelerator. The latency results for each of the non-weight-shared, weight-shared, and weight-shared-with-PASM accelerators is obtained from Vivado_HLS Synthesis reports and the percentage differences graphed as seen in Figure 3-12. The latency of the weight-shared-with-PASM in Figure 3-12 was between 8.5% for 4-bin and 12.75% for 16-bin greater than that of the corresponding weight-shared version, which is expected due to the indirection of the PAS units.

Latency can be further reduced by relaxing the *ALLOCATION* directive (see line 3 of Listing 3.2) constraint on the multiplier. If more post-pass multipliers are used, then the latency drops with a corresponding increase in power and area, which may be acceptable depending on target device resources available.

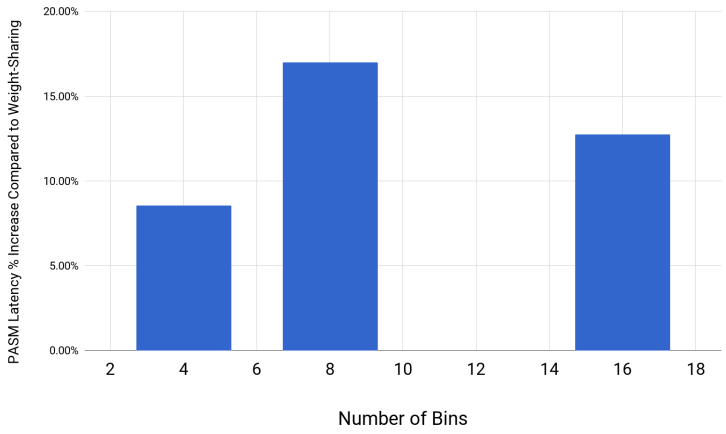
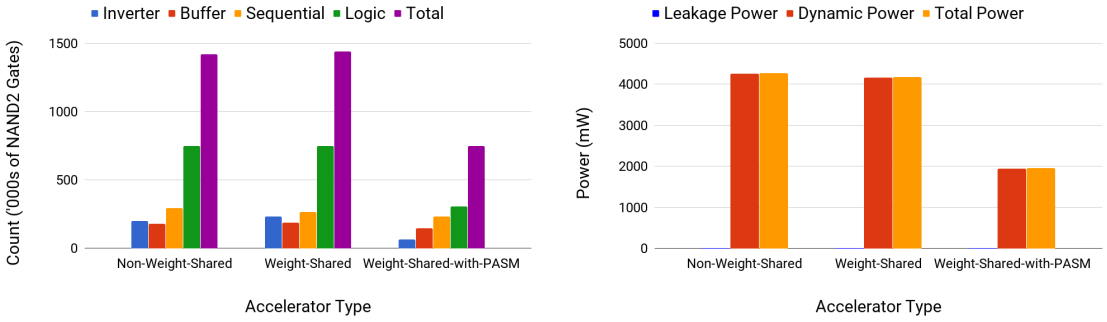


Figure 3-12: **Latency of weight-shared-with-PASM convolution compared to weight-shared convolution.**



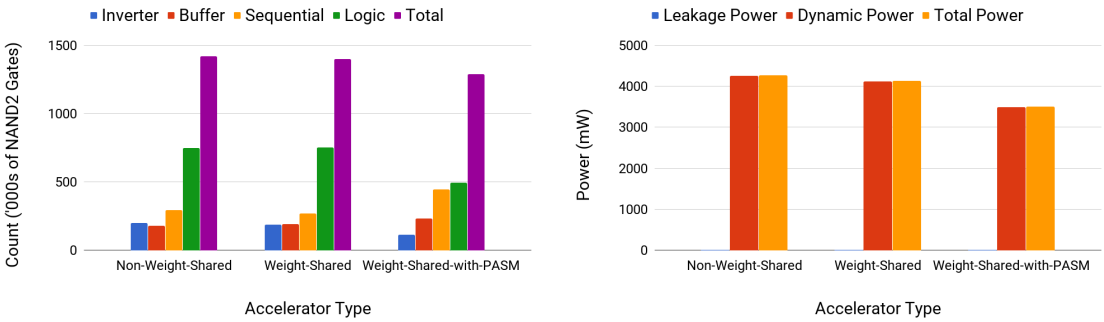
(a) ASIC Gate count for 32-bit kernel, 4-bin accelerators.

(b) ASIC Power consumption for 32-bit kernel, 4-bin accelerators.

Figure 3-13: 4-bin, 32-bit Kernel Weight-shared-with-PASM vs. Weight-shared Gate Count and Power Comparisons in ASIC.

For a 4-bin PASM accelerator, with 32-bit wide kernels, Figure 3-13a shows the gate count reports obtained from Cadence "report gates" command and normalised to a NAND2 gate. PASM uses 47.2% fewer total NAND2 gates compared with the non-weight-shared version and 47.8% fewer total NAND2 gates compared with weight-shared design. Figure 3-13b obtained from Cadence "report power" command, PASM uses 54.3% less total power when compared with its non-weight-sharing counterpart and 53.2% less total power when compared with the weight-shared version.

For an 8-bin, 32-bit wide kernel PASM accelerator, Figure 3-14a obtained from Cadence "report gates" command and normalised to a NAND2 gate, PASM uses



(a) ASIC Gate count for 32-bit kernel, 8-bin accelerators.

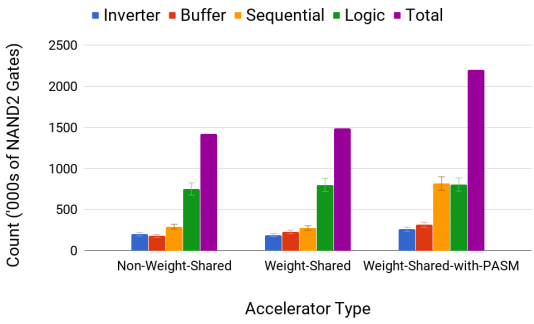
(b) ASIC Power consumption for 32-bit kernel, 8-bin accelerators.

Figure 3-14: 8-bin, 32-bit Kernel Weight-shared-with-PASM vs. Weight-shared Gate Count and Power Comparisons in ASIC.

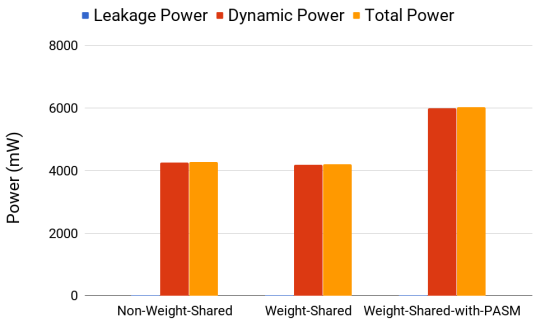
9.4% fewer total NAND2 gates compared with the non-weight-shared and 8.1% fewer total NAND2 gates compared with the weight-shared accelerators. Figure 3-14b obtained from Cadence "report power" command, PASM consumes 18.1% less total power when compared with its non-weight-sharing and 15.2% less total power when compared with the weight-sharing accelerator.

For a 16-bin, 32-bit wide weight-shared-with-PASM accelerator, PASM no longer offers a good return with this level of unrolling, pipelining and partitioning of the ***IFMBin***, at least when targeted at a 1GHz ASIC with this 45nm process cell library as it uses more NAND2 gates (see Figure 3-15a) and power (see Figure 3-15b) compared with the weight-shared accelerator. This is due to the ASIC tools increasing the area and the power to meet timing at 1GHz for the 16-bins at 32-bit wide PASM. To achieve better power and area results for PASM at 16-bins or greater, it might be better to target a lower clock frequency, for example, 800MHz. Alternatively, use a more efficient geometry ASIC cell library. Design changes could be made to reduce pipelining and unrolling of the levels of the inner four of the *for* loops of the convolutional code. The *for* loop changes would reduce the area and power while making it easier for the ASIC tools to achieve timing. However, this may increase the latency of the accelerator.

Due to the increased academic and industrial interest in applying INT8 approximations to reduce memory storage and bandwidth of the kernel data [Dettmers 2015; Fu et al. 2016], we show the results for the 8-bit kernel versions of the accel-

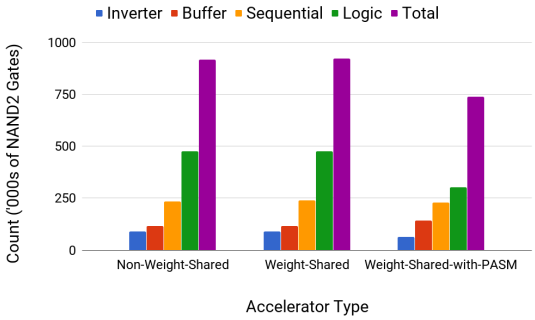


(a) ASIC Gate count for 32-bit kernel, 16-bin accelerators.

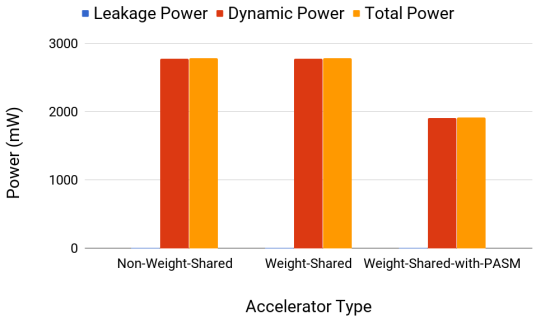


(b) ASIC Power consumption for 32-bit kernel, 16-bin accelerators.

Figure 3-15: 16-bin, 32-bit Kernel Weight-shared-with-PASM vs. Weight-shared Gate Count and Power Comparisons in ASIC.



(a) ASIC Gate count for INT8-bit kernel, 4-bin accelerators.



(b) ASIC power consumption for INT8-bit kernel, 4-bin accelerators.

Figure 3-16: 4-bin, INT8-bit Kernel Weight-shared-with-PASM vs. Weight-shared Gate Count and Power Comparisons in ASIC.

erators with 4-bins. This demonstrates that for a bin depth of 4, PASM achieves a 19.8% reduction in gate count. Figure 3-16a obtained from Cadence "report gates" command is normalised to a NAND2 gate and a 31.3% reduction in power compared to the weight-sharing version, Figure 3-16b also obtained from Cadence "report power" command.

3.5.2 FPGA Results

We implement the weight-shared-with-PASM accelerator in the Xilinx 7-series Zynq FPGA, the XC7Z045 part implemented on the Zynq ZC706 development board. Timing constraints in Xilinx design constraint (XDC) are created such that the accelerator designs met timing at 200MHz. The resets are set as active high synchronous

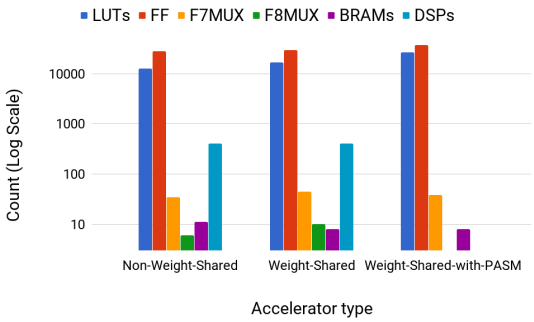
resets for better FPGA power performance. The state machines are set to grey encoding.

The ***IFM***, ***IFMBin***, and ***kernel*** were cached in BRAM in the FPGA. This allows for larger cache storage of IFM and weight values and further reduce the power and area of the accelerator. However, a larger IFM and kernel cache could be employed for greater throughput of the accelerators, but for comparison with the ASIC implementation, the same IFM and kernel dimensions are used.

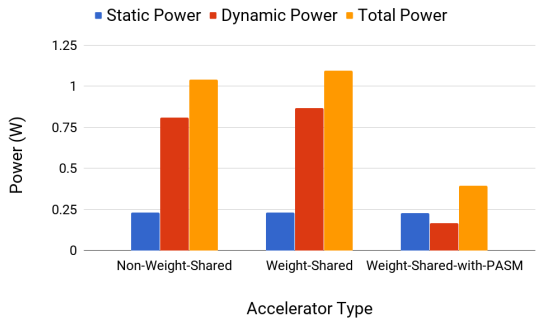
When using the *UNROLL* and *PIPELINE* directives with the *for* loops and using Vivado_HLS synthesis followed by RTL synthesising and fully implementing the designs with Vivado, the non-weight-shared and weight-shared versions of the 16-bin, 32-bit kernel data designs utilises 405 DSP units on the FPGA of the ZC706 board. If a smaller, more resource-constrained FPGA is required for cost reasons, like the Xilinx XC7Z020 part found on the Xilinx PYNQ-Z1 low-cost development board, then the non-weight-shared and weight-shared versions of the design would over utilise the 220 DSP units of the PYNQ-Z1 board's XC7Z020 FPGA part.

The weight-shared-with-PASM version of the design for the same 4-bin, 32-bit kernel, Figure 3-17a obtained with Vivado's "report_utilization" command, similarly unrolled and pipelined, HLS synthesised in Vivado_HLS followed by RTL synthesised and fully implemented in Vivado, only utilises 3 DSP units, 99% fewer DSPs than the other versions of the accelerator with the same 12% increase in latency as the ASIC implementation. PASM also consumes 28% fewer BRAMs, while consuming 64% less power than the weight-shared accelerator, Figure 3-17b, obtained with Vivado's "report_power" command. Increasing the number of post-pass MACs decreases the latency slightly while increasing the power consumption and DSP usage. The bottleneck is in the accumulators of the PAS which can be defined by Equation 2 and must be larger than that of B-bins for PASM to be effective, and area, and energy efficient, as seen in Table 3.1 and Table 3.2.

For an 8-bin PASM accelerator, with 32-bit kernels, Figure 3-18a obtained with Vivado's "report_utilization" command, PASM uses 99% fewer DSPs and 28% fewer BRAMs compared with the weight-shared design. Figure 3-18b obtained with Vi-

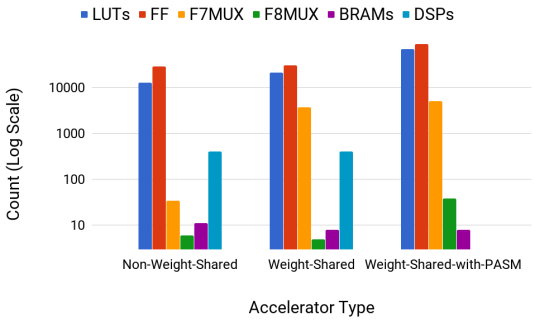


(a) Cell count for 32-bit kernel, 4-bin accelerators.

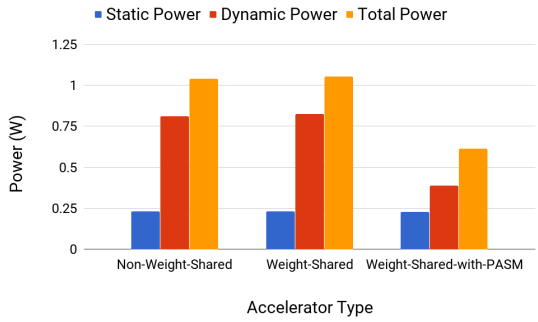


(b) Power consumption for 32-bit kernel, 4-bin accelerators.

Figure 3-17: 4-bin, 32-bit Kernel Weight-shared-with-PASM vs. Weight-shared Gate Count and Power Comparisons in FPGA.



(a) Cell count for 32-bit kernel, 8-bin accelerators.



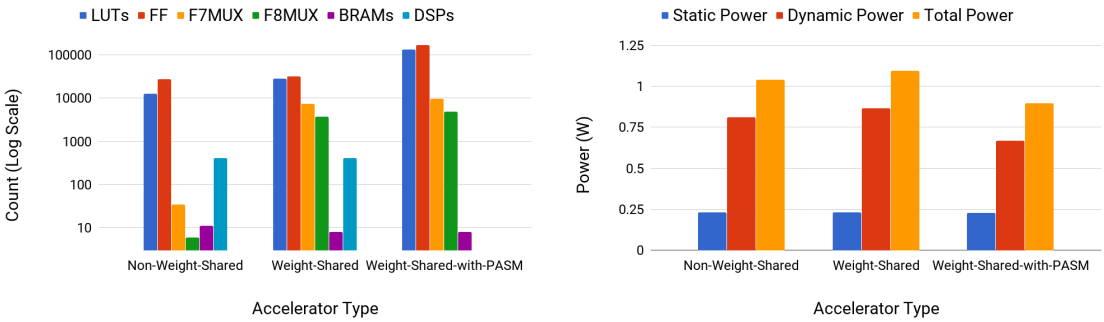
(b) Power consumption for 32-bit kernel, 8-bin accelerators.

Figure 3-18: 8-bin, 32-bit Kernel Weight-shared-with-PASM vs. Weight-shared Gate Count and Power Comparisons in FPGA.

vado's "report_power" command, PASM uses 41.6% less total power when compared with its weight-shared version.

For a 16-bin PASM accelerator, with 32-bit kernels the utilisation reported with Vivado's "report_utilization" command, Figure 3-19a, PASM uses 99% fewer DSPs and 28% fewer BRAMs compared with the weight-shared design. Figure 3-19b, PASM uses 18% less total power when compared with its weight-shared version, reported with Vivado's "report_power" command.

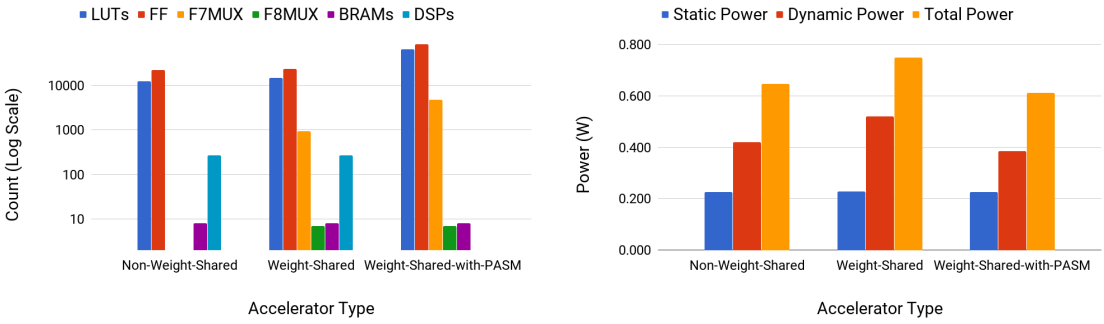
It is also possible to clock the PASM at higher clock speeds for the same latency than that of the weight-shared counterpart, but again for the sake of comparison, clock speeds are kept consistent between all versions of the accelerators.



(a) Cell count for 32-bit kernel, 16-bin accelerators.

(b) Power consumption for 32-bit kernel, 16-bin accelerators.

Figure 3-19: 16-bin, 32-bit Kernel Weight-shared-with-PASM vs. Weight-shared Gate Count and Power Comparisons in FPGA.



(a) Cell count for INT8-bit kernel, 8-bin accelerators.

(b) Power consumption for INT8-bit kernel, 8-bin accelerators.

Figure 3-20: 8-bin, INT8-bit Kernel Weight-shared-with-PASM vs. Weight-shared Gate Count and Power Comparisons in FPGA.

If INT8 approximations are desired for the weight data, an 8-bit wide, 8-bin PASM accelerator, Figure 3-20a again obtained with Vivado's "report_utilization" command, uses 99% fewer DSPs but the same number of BRAMs as it's weight-shared counterpart. Figure 3-20b, PASM uses 18.3% less total power when compared with its weight-shared version.

For a 16-bin, 8-bit wide PASM accelerator, PASM no longer offers a good return when targeted at a 200MHz FPGA with this level of unrolling, pipelining and partitioning of the ***IFMBin*** as it uses more flip-flop gates and power, exceeding the gate count and power of the DSP units being used in the weight-shared accelerator. At this stage, it would be better to either implement the ***IFMBin*** in dual port BRAM

and incur a slight increase in latency or do not unroll and pipeline as many levels of the inner four of the *for* loops of the convolutional code.

3.5.3 Overall Results

The precision of the results of a weight-shared CNN accelerator that uses PASM are identical to that of a weight-shared CNN accelerator using traditional MACs. The same filters and IFM data are being used for the weight-shared accelerator as demonstrated in Figure 3-3 and the weight-shared-with-PASM accelerator shown in Figure 3-5. While PASM has a different underlying process of permuting the convolution, the results of a convolution layer are identical to that of a standard MAC weight-shared accelerator, except PASM adds a 12.5% increase in latency in obtaining the result but with vastly reduced power consumption and area (NAND2 gates) compared to the traditional MAC version.

As suggested in Han *et al.*, [2016b], they show that the Top-5 classification accuracy of their weight-shared CNN accelerator is 19.70% compared to 19.73% Top-5 classification accuracy of the baseline non-weight-shared CNN accelerator due to there being many less filter weight values. When PASM is used in a weight-shared CNN accelerator, the classification accuracy is unaffected when compared to the baseline weight-shared CNN accelerator counterpart as the same filter weight values of the weight-shared CNN accelerator are used, and the same output feature map results are obtained.

PASM is beneficial for up to sixteen weight bins and 32-bits for FPGA at an iso-frequency of 200MHz and eight weight bins and 32-bits for ASIC at an iso-frequency of 1GHz on a 45nm process when coded using SystemC with the above unrolling, pipelining and partitioning configuration. As demonstrated earlier in the chapter, were a weight-shared-with-PASM CNN accelerator to be coded in Verilog, the numbers of bins supported could indeed be higher. We wanted to experiment with differing pipelining, unrolling and partitioning directives and their effect on making PASM more efficient, something which would have been impractical had it been coded in Verilog, so SystemC was used. Further SystemC and other SRAM optimisations (for IFM and output feature map caching) could have been done to the

accelerators, but this was not the focus of this chapter and maybe undertaken as future work.

3.6 Conclusion

As discussed in section 2.1, ASICs and FPGAs are often used to hardware accelerate the convolution layers of a CNN where up to 90% of the computation time is consumed. This computation requires large amounts of multipliers as part of the many thousands of MAC operations needed in the convolution layer. These multipliers consume large amounts of physical and computational IC die resources or DSP units on a FPGA. Hardware accelerators have been proposed that reduced the amount of kernel data required by the neural network by dictionary compressing the weight values after training the network. This "weight-sharing" reduces the bandwidth and power of the data transfers from external memory but still requires large numbers MAC units.

We reduce power and area of the CNN accelerator by implementing PASM in a weight-shared CNN accelerator. PASM re-architects the MAC to count the frequency of each weight instead and place it in a bin. The accumulated value is computed in a subsequent multiply phase, significantly reducing gate count and power consumption of the CNN. We code in Verilog a 16-MAC weight-shared accelerator and a 16-PAS-4-MAC weight-shared-with-PASM accelerator and compare the logic resource requirements of a $b = 16$ bin for varying w-bit widths. Gate counts are normalised to a NAND2X1 gate. For $w = 32$-bit wide the 16-PAS-4-MAC has overall 66% fewer logic gates and consumes 70% less total power than the 16-MAC.

To further evaluate the efficiency gains of PASM, we implement PASM in a weight-sharing CNN accelerator. We compare it to a non-weight-shared accelerator and a weight-shared accelerator, targeted at a 1GHz 45nm ASIC. The gate count area and power consumption for the weight-shared-with-PASM is lower compared to the weight-shared version. For a 4-bin weight-shared-with-PASM accelerator that accepts a 5×5 IFM with a 3×3 kernel and 15 input channels and 2 output channels, an ASIC implementation of PASM saves 48% NAND2 gates and 53.2% power when compared to its weight-shared counterpart, with only a 12% increase in latency.

We show that the weight-shared-with-PASM accelerator can be implemented in a resource-constrained FPGA. For an accelerator with the same dimensions as the ASIC version implemented on the FPGA to run at 200MHz, PASM uses 99% fewer DSPs and 28% fewer BRAMs compared with the weight-shared design. For 16-bin PASM, PASM uses 18% less total power when compared with its weight-shared version.

Even if INT8 approximations are desired for the weight data, an 8-bit wide, 4-bin PASM accelerator running at 200MHz on the FPGA uses 99% fewer DSPs and 28% fewer BRAMs compared with the weight-shared design. An INT8 operation PASM also uses 47% less total power when compared with its INT8 weight-shared version.

Quantisation and weight-sharing neural networks are active research areas, particularly for reducing DRAM bus bandwidth usage and in applications such as RNNs and LSTM networks. Weight-sharing allows for the implementation of a CNN in small, low power embedded systems as less RAM is required to store the weight values. Weight-sharing also offers a more rapid way of implementing an inference network on a small memory embedded device without the large training phase required of, say a BNN. Weight-sharing is used in other types of networks such as regional-CNNs, RNNs and LSTMs so PASM may be a good fit there too. Wherever the number of shared weights is sufficiently small, PASM units may be an attractive alternative to a conventional weight-sharing MAC unit.

The clock frequency verses the concurrency of PASM was not investigated. Future work might investigate what energy efficiencies could be made if different implementation clock frequencies are used at different layers of the CNN.

PASM could be used in any shared binned data system that might be used in other arenas, such as GPU acceleration. However, this has not be investigated or measured in this work but might be an avenue for future investigation.

4 Hardware Optimised Bit-sliced Floating-Point Operators (HOBFLOPS) for CNNs

4.1 Introduction

MANY researchers have shown that CNN inference is possible with low-precision integer [Jouppi et al. 2017] and floating-point (FP) [Chung et al. 2018; Fowers et al. 2018] arithmetic (see section 2.1 beginning on page 9 for more details). Almost all processors provide support for 8-bit integers, but not for bit-level custom-precision FP types, such as 9-bit FP. Typically processors support a small number of relatively high-precision FP types, such as 32- and 64-bit [Intel 2020]. However, there are good reasons why we might want to implement custom-precision FP on regular processors. Researchers and hardware developers may want to prototype different levels of custom FP precision that might be used for arithmetic in CNN accelerators [Zaidy 2016; Gupta et al. 2015; DiCecco et al. 2017]. Furthermore, fast custom-precision FP CNNs in software may be valuable, particularly in cases where memory bandwidth is limited.

To address custom-fp in software of CPUs, FP simulators, such as the Flexfloat [Tagliavini et al. 2018], and Berkeley's SoftFP [Hauser 1999], are available. These simulators support arbitrary or custom range and precision FP such as 16-, 32-, 64-, 80- and 128-bit with corresponding fixed-width mantissa and exponents respectively. However, the simulators' computational performance may not be sufficient for the requirements of high throughput, low latency arithmetic systems such as CNN convolution.

We propose hardware optimized bitslice-parallel floating-point operators (HOBFLOPS). HOBFLOPS generates FP units, using software *bitslice parallel* arithmetic, efficiently emulating FP arithmetic at arbitrary mantissa and exponent bit-widths on processors that do not otherwise contain low-precision FP. We exploit bit-slice parallelism techniques to pack the SIMD vector registers of the microprocessor more efficiently. Also, we exploit bitwise logic optimisation strategies of a commercial hardware synthesis tool to optimise the associated bitwise arithmetic. A source-to-source generator converts the netlists to the target processor's bitwise SIMD vector operators. We propose HOBFLOPS as low-precision FP arithmetic in the MAC offers increased classification accuracy compared with INT8 or other number representation counterparts. As we will see, HOBFLOPS saves machine operations for a comparable convolution compared to its counterparts.

To evaluate performance we benchmark HOBFLOPS8 through to HOBFLOPS16e parallel MACs against arbitrary-precision Flexfloat, and 16- and 32-bit Berkeley's SoftFP, implemented in CNN convolution with Arm and Intel scalar and vector bitwise instructions. We show HOBFLOPS offers significant performance boosts compared to Flexfloat, and SoftFP. We show that our software bitslice parallel FP is both more computationally efficient and offers greater bit-level customisability than other software FP emulators.

We make the following contributions:

- We present a full design flow from a VHDL FP core generator to arbitrary-precision software bitslice parallel FP operators. HOBFLOPS are optimised using hardware design tools and logic cell libraries, and domain-specific code generator.
- We demonstrate how 3-input Arm NEON bitwise instructions *e.g.*, SEL (multiplexer), and AVX512 bitwise ternary operations are used in standard cell libraries to improve the efficiency of the generated code.
- We present an algorithm for implementing CNN convolution with the very wide vectors that arise in bitslice parallel vector arithmetic.
- We evaluate HOBFLOPS on Arm Neon and Intel AVX2 and AVX512 processors. We find HOBFLOPS achieves approximately $3\times$, $5\times$, and $10\times$ the performance of Flexfloat respectively.

- We evaluate various widths of HOBFLOPS from HOBFLOPS8–HOBFLOPS16e. We find *e.g.*, HOBFLOPS9 outperforms Flexfloat 9-bit by $7\times$ on Intel AVX512, by $3\times$ on Intel AVX2, and around $2\times$ Arm Neon. The increased performance is due to:
  - Bitslice parallelism of the very wide vectorisation of the MACs of the CNN;
  - Our efficient code generation flow.

The rest of this article is organised as follows. Section 4.2 highlights our motivation and gives background on other CNN accelerators use of low-precision arithmetic types. Section 4.3 outlines bitslice parallel operations and introduces HOBFLOPS, shows the design flow, types supported and how to implement arbitrary-precision HOBFLOPS FP arithmetic in a convolution layer of a CNN. Section 4.4 shows significant increases in performance of HOBFLOPS8–HOBFLOPS16e compared with Flexfloat, and SoftFP on Intel's AVX2 and AVX512 and Arm Neon processors. We outline related work in Section 4.5 and conclude with Section 4.6.

4.2 Background and Motivation

Arbitrary precision floating-point computation is largely unavailable in CPUs. Soft FP simulation is available but typically lacks the computation performance required of low latency applications. Researchers often reduce the precision to a defined fixed-point basis for CNN inference, potentially impacting CNN classification accuracy [Lo et al. 2018].

Reduced-precision CNN inference, particularly CNN weight data, reduces computational requirements due to memory accesses, which dominate energy consumption. Energy and area costs are also reduced in ASICs and FPGAs [Sze et al. 2017].

Johnson [Johnson 2018] suggests that little effort has been made in improving FP efficiency so proposes an alternative floating-point representation. They show that a 16-bit log float multiply-add is $0.68\times$ the IC die area compared with an IEEE-754 float16 fused multiply-add, while maintaining the same significand precision and dynamic range. They also show that their reduced FP bit precision compared

to float16 exhibits $5\times$ power saving. We investigate if similar efficiencies can be mapped into software using a hardware tool optimisation flow.

Kang *et al.*, [Kang 2018] investigate short, reduced FP representations that do not support not-a-numbers (NANs) and infinities. They show that shortening the width of the exponent and mantissa reduces the computational complexity within the multiplier logic. They compare fixed point integer representations with varying widths up to 8-bits of their short FP in various CNNs, and show around a 1% drop in classification accuracy, with more than 60% reduction in ASIC implementation area. Their work stops at the byte boundary, leaving us to investigate other arbitrary ranges.

Researchers often use custom precision in the design of arithmetic accelerators implemented in hardware such as FPGA or ASIC as discussed in section 2.1 on page 9. Microsoft proposes MS-FP8 and MS-FP9, which are 8-bit and 9-bit FP arithmetic that they exploit in a quantised CNN [Chung et al. 2018; Fowers et al. 2018]. Microsoft alters the Minifloat 8-bit that follows the IEEE-754 specification (1-sign bit, 4-exponent bits, 3-mantissa bits) [IEEE 2019]. They create MS-FP8, of 1-sign bit, 5-exponent bits, and 2-mantissa bits. MS-FP8 gives a larger representative range due to the extra exponent bit but lower precision than Minifloat, caused by the reduced mantissa. MS-FP8 more than doubles the performance compared to 8-bit integer operations, with negligible accuracy loss compared to full float. To improve the precision, they propose MS-FP9, which increases the mantissa to 3 bits and keeps the exponent at 5 bits. Their later work [Fowers et al. 2018] uses a shared exponent with their proposed MS-FP8 / MS-FP9, *i.e.*, one exponent pair used for many mantissae, sharing the reduced mantissa multipliers. We do not investigate shared exponent. Their work remains at 8- and 9-bit for FPGA implementation and leaves us to investigate other bit-precision and ranges.

Rzayev *et al.'s*, Deep Recon work [Rzayev et al. 2017] analyses the computation costs of DNNs. They propose a reconfigurable architecture to efficiently utilise computation and storage resources, thus allowing DNN acceleration. They pay particular attention to comparing the prediction error of three CNNs with different fixed and FP precision. They demonstrate that FP precision on the three CNNs is 1-bit more efficient than fixed bit-width. They also show that the 8-bit FP is around $7\times$

more energy-efficient and approximately $6\times$ more area efficient than 8-bit fixed precision. We further the area efficiency investigation.

Existing methods of emulating FP arithmetic in software primarily use existing integer instructions to implement the FP computation steps. This works well for large, regular-sized FP types such as FP16, FP32, or FP64. Berkeley offer SoftFP emulation [Hauser 1999] for use where, *e.g.*, only integer precision instructions are available. SoftFP emulation supports 16- to 128-bit arithmetic and does not support low bit-width custom precision FP arithmetic or parallel arithmetic.

The Flexfloat `C++` library of Tagliavini *et al.*, [Tagliavini et al. 2018], offers alternative FP formats with variable bit-width mantissa and exponents. They demonstrate Flexfloat is up to $2.8\times$ and $2.4\times$ faster than MPFR and SoftFloat, respectively for various benchmarks. They do not explore arbitrary bit-precision or other optimisation techniques on the proposed number formats, which our work does. Both Flexfloat and SoftFP simulators operate at a much lower performance than that of the hardware floating-point unit (FPU).

Other researchers investigate optimising different representations of FP arithmetic. Xu *et al.*, [Xu and Gregg 2017] propose bitslice parallel arithmetic and present FP calculations undertaken on a fixed point unit. Instead of storing vectors in the traditional sense of storing 17-bit vectors inefficiently in a 32-bit register, they instead store thirty-two 17-bit words transformed into bitslice parallel format. Xu *et al.*, manually construct bitwise arithmetic routines to perform integer or FP arithmetic, while the vectors remain in a bitslice parallel format. When coded in C/C++ and AVX2 SIMD instructions, they demonstrate this approach is efficient for low-precision vectors, such as 9-bit or 11-bit arbitrary FP types. We investigate beyond 11-bit and AVX2 implementation by generating and optimising the process using ASIC design flow tools and applying HOBFLOPS to CNN convolution on Arm Neon, Intel AVX2 and AVX512.

Researchers investigate different bit precision and representations of the FP number base. Google's TPU ASIC [Jouppi et al. 2017] implements bfloat16 [Google 2019], a 16-bit truncated IEEE-754 FP single-precision format. Bfloat16 preserves dynamic range of the 32-bit format due to the 8-bit exponent. The precision is reduced in the mantissa from IEEE's 24-bits down to 7-bits.

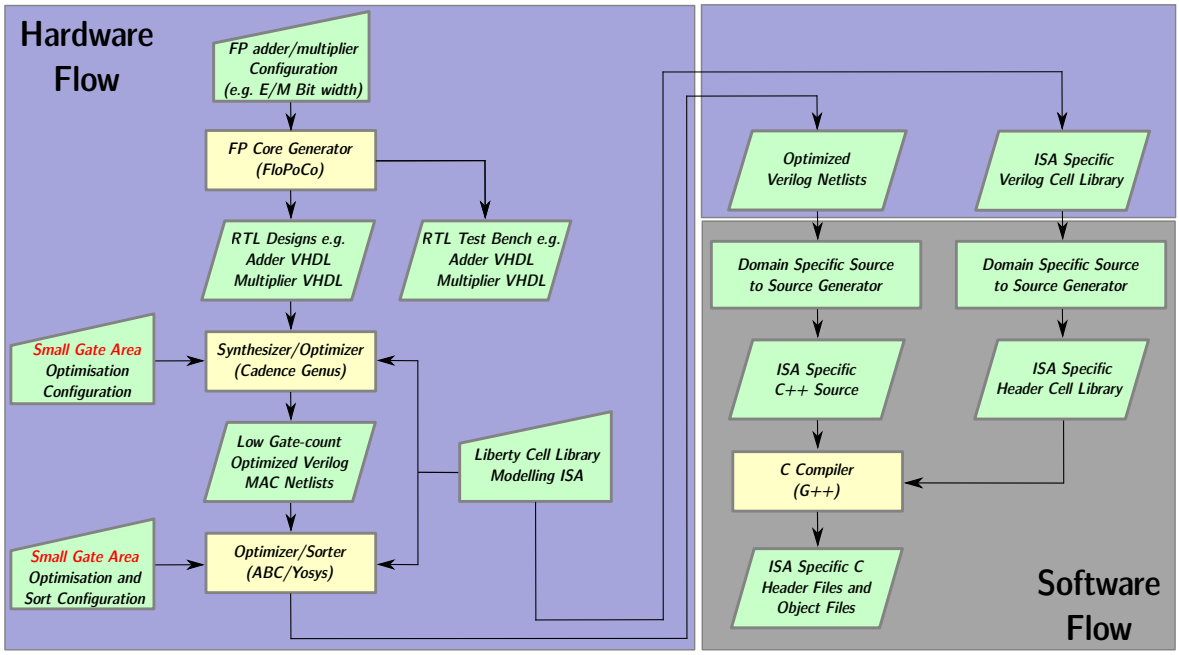


Figure 4-1: **Flow for Creating HOBFLOPS Bitwise Intrinsic Operations. Yellow signifies third party tools. We start in the hardware domain and cross into the software domain.**

Nvidia has proposed the new tensor float 32 (TP32) number representation [Choquette et al. 2021], which is a sign bit, 8-bit exponent and 10-bit mantissa. This 19-bit floating-point format is an input format which truncates the IEEE FP32 mantissa to 10-bits. TP32 still produces 32-bit floating-point results to move and store in the GPU. Similar to Google, Nvidia does not investigate other bit-widths of FP range and precision, something our work demonstrates.

4.3 Approach

In this section, we present how to produce HOBFLOPS arithmetic units. HOBFLOPS generates efficient software emulation parallel FP arithmetic units optimised using hardware synthesis tools. HOBFLOPS investigates reduced complexity FP, [Kang 2018; Johnson 2018] that is more efficient than fixed-point [Rzayev et al. 2017]. HOBFLOPS considers alternative FP formats [Tagliavini et al. 2018] and register packing with bit-sliced arithmetic [Xu and Gregg 2017]. We demonstrate HOBFLOPS in a CNN convolution layer.

Figure 4-1 outlines our flow for creating HOBFLOPS arithmetic units. We generate the RTL representations of arbitrary-precision FP multipliers and adders using the FP unit generator, FloPoCo [De Dinechin et al. 2009]. We configure the FP

```
flopoco FPMult pipeline=0 useHardAdd=0 frequency=1 plainVHDL=1 target=Virtex6 wE=5 wF
    =2 wFOut=3 outputFile=FPMult_5_2_3.vhd
flopoco FPMult pipeline=0 useHardAdd=0 frequency=1 plainVHDL=1 target=Virtex6 wE=5 wF
    =2 wFOut=5 outputFile=FPMult_5_2_5.vhd
flopoco FPAdd  pipeline=0 useHardAdd=0 frequency=1 plainVHDL=1 target=Virtex6 wE=5 wF
    =3 outputFile=fpAdd_5_3.vhd
flopoco FPAdd  pipeline=0 useHardAdd=0 frequency=1 plainVHDL=1 target=Virtex6 wE=5 wF
    =5 outputFile=fpAdd_5_5.vhd
```

Listing 4.1: Example FloPoCo Script for standard and extended precision Multipliers and Adders.

```
library(avx512) {
cell (LUT000X1) {
area : 1.0;
  cell_leakage_power : 0.0;
  pin(Y)  {
    direction : output;
    capacitance : 0;
    rise_capacitance : 0;
    fall_capacitance : 0;
    max_capacitance : 0;
    function : "0";
  }
}
}
```

Listing 4.2: Snippet Showing One Cell of AVX512 Standard Cell Library.

range and precision of HOBFLOPS adders and multipliers, Listing 4.1. Note the option *frequency=1* is set so FloPoCo relaxes timing constraints to produce the smallest asynchronous gate count design. The low number of gates will 1-1 map to bitwise logic SIMD vector instructions.

We produce standard cell libraries of logic gate cells mapped to bitwise logic instructions of the target microprocessor architecture. For example, AND, OR, XOR, the SEL (multiplexer) bitwise instructions are supported on Arm Neon. The 256-ternary logic LUT bitwise instructions of the Intel AVX512 are supported. NOTE: due to the lack of availability of Arm Scalable Vector Extension (SVE) devices, we show results for an Arm Neon device. The same approach can be applied to creating a cell library for an Arm SVE device or other SIMD vector instruction processor, when available.

We use the ASIC tool, Cadence Genus with our standard cell libraries, Listing 4.2, and small gate area optimisation script, Listing 4.3. Genus synthesises the adders and multipliers into Verilog netlists. The synthesis and technology mapping suite, Yosys ASIC [Wolf et al. 2013] and ABC optimiser [Brayton and Mishchenko 2010],

```
proc runFlow {}  {
  read_libs cellLib/avx.lib
  read_hdl -library work -f filesList.txt
  set top FPMult_5_2_5_2_5_3
  elaborate
  ungroup -all -flatten -force
  syn_generic -effort high
  syn_map -effort high
  syn_opt -effort high
  set netsList [get_nets]
  sort_collection $netsList name
  writeReps reports netlists $top
}
proc writeReps {_repsDir_ _netsDir_ _top_} {
  report gates >  ${_repsDir_}/${_top_}_gates.rpt
  report area [get_designs $_top_] > ${_repsDir_}/${_top_}_area.rpt
  write_hdl -mapped > ${_netsDir_}/$_top_.v
  write_sdc > ${_netsDir_}/constraints.sdc
}
file mkdir reports
file mkdir netlists
runFlow
```

Listing 4.3: Example Cadence Genus Script.

```
read_liberty -lib ../cellLib/avx2.lib
read_verilog FPMult_5_2_5_2_5_3_comb_uid2.v
hierarchy -check -top FPMult_5_2_5_2_5_3_comb_uid2
synth -top FPMult_5_2_5_2_5_3_comb_uid2
abc -liberty ../cellLib/avx2.lib
proc; opt; flatten; proc; opt; pmuxtree; opt; memory -nomap; opt; clean;
techmap; opt
clean
torder -noautostop
write_verilog FPMult_5_2_5_2_5_3_comb_uid2Topo.v
```

Listing 4.4: Example Yosys-ABC script.

Listing 4.4, allows further optimisation and topologically sort of the netlists with the same cell libraries.

Our custom domain-specific source-to-source generator, Listing 4.5, converts the topologically sorted netlists into a C/C++ header of bitwise operations. In parallel,

```
cd $1/netlists
for i in $(find . -name "*.v");
do
  TOP_MODULE=$(basename $i .v)
  cp template.h ${TOP_MODULE}.h
  sed -i "s/TEMPLATE_H_/${TOP_MODULE}_H_/g" ${TOP_MODULE}.h
  sed -i "s/void template/void ${TOP_MODULE}/g" ${TOP_MODULE}.h
  sed -e '1,11d' < ${TOP_MODULE}.v > ${TOP_MODULE}_1.h
  sed -i 's/wire/BSFP_T/g' ${TOP_MODULE}_1.h
  sed -i "s/.A (/\&/g" ${TOP_MODULE}_1.h
  sed -i "s/.B (/\&/g" ${TOP_MODULE}_1.h
  sed -i "s/.C (/\&/g" ${TOP_MODULE}_1.h
  sed -i "s/.Y (/\&/g" ${TOP_MODULE}_1.h
  sed -i "s/),/,/g" ${TOP_MODULE}_1.h
  sed -i "s/));/);/g" ${TOP_MODULE}_1.h
  sed -i "s/ g.*(/(/g" ${TOP_MODULE}_1.h
  sed -i "s/endmodule/\}/" ${TOP_MODULE}_1.h
  sed -e "\$a#endif  \/\/ ${TOP_MODULE}_H_" ${TOP_MODULE}_1.h > ${TOP_MODULE}_2.h
  cat ${TOP_MODULE}.h ${TOP_MODULE}_2.h > ${TOP_MODULE}_3.h
  mv ${TOP_MODULE}_3.h ${TOP_MODULE}.h
  rm ${TOP_MODULE}_1.h ${TOP_MODULE}_2.h
done
```

Listing 4.5: Example Source-to-Source Generator.

```
void AND2X1(u256 *A, u256 *B, u256 *Y) {
  *Y = _mm256_and_si256(*A, *B);}
void NOTX1(u256 *A, u256 *Y) {
  // Inverter could be implemented in many ways
  *Y = _mm256_xor_si256(*A, _mm256_set1_epi32(-1));}
void OR2X1(u256 *A, u256 *B, u256 *Y) {
  *Y = _mm256_or_si256(*A, *B);}
void XOR2X1(u256 *A, u256 *B, u256 *Y) {
  *Y = _mm256_xor_si256(*A, *B);}
void ANDNOT2X1(u256 *A, u256 *B, u256 *Y) {
  *Y = _mm256_andnot_si256(*A, *B);}
```

Listing 4.6: Macros for AVX2 Cell Library Bitwise Operator Definitions

Table 4.1: HOBFLOPS Cell Libraries' Support for Arm Neon and Intel Intrinsic Bitwise Logic Operations.

Arm (64-bit)		**Arm Neon (128-bit) [Arm 2019]**		**Intel (64-bit)**		**Intel AVX2 (128-, 256-bit)**		**Intel AVX512 (512-bit) [Intel 2020]**	
AND	A & B	AND	A & B	AND	A & B	AND	A & B	LUT000	0
OR	A | B	OR	A | B	OR	A | B	OR	A | B	LUT001	(A | (B | C)) ^ 1
XOR	A ^ B	XOR	A ^ B	XOR	A ^ B	XOR	A ^ B	LUT002	~ (B | A) C
NOT	~A	NOT	~A	NOT	~A	NOT	~A	LUT003	(B | A) ^ 1
ORN	A & (~B)	ORN	A | ~B			ANDNOT	~A & B	LUT004	~(A | C) B
		SEL	(~((S & A) | (~S & B)))					LUT005	(C | A) ^1
								...	*... (truncated)*
								LUT253	A | (B | (C ^ 1))
								LUT254	A | (B | C)
								LUT255	1

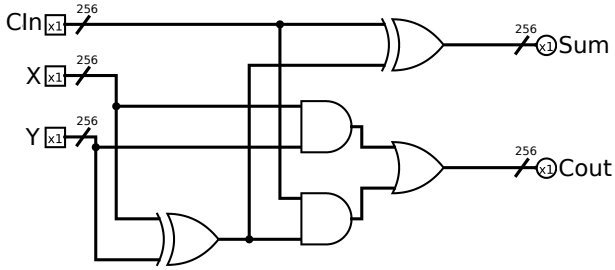
the generator converts the standard cell libraries into equivalent C/C++ cell library headers of SIMD vector extension instructions of the target processor ISA.

We create a CNN convolution layer to include the HOBFLOPS adder, multiplier and cell library headers corresponding to the target ISA, e.g., Listing 4.6, and compile with G++.

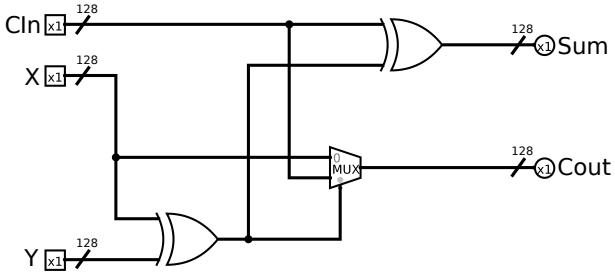
4.3.1 HOBFLOPS Cell Libraries for Arm Neon, Intel AVX2 and AVX512

We create three Liberty standard cell libraries mapping hardware bitwise representations to Arm Neon Intel X86_64, AVX, AVX2 and AVX512 SIMD vector intrinsics, architectures which were readily available to us. Other processors such as PowerPC and RiscV can be targetted as long as the target processor contains SIMD bitwise vector instructions that can be modelled in the hardware cell library.

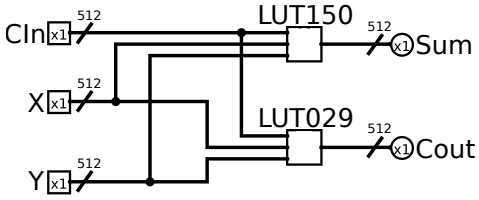
The Arm Neon SEL (multiplexer) bitwise multiplexer instruction is a 3-input bitwise logic instruction, whereas all other Neon bitwise logic instructions modelled in the cell library are 2-input. The ternary logic LUT of the Intel AVX512 is a 3-input bitwise logic instruction that can implement 3-input Boolean functions. An 8-bit immediate operand to this instruction specifies which of the 256 3-input functions



(a) **Intel AVX2 Bitwise Operations.**



(b) **Arm Neon Bitwise Operations.**



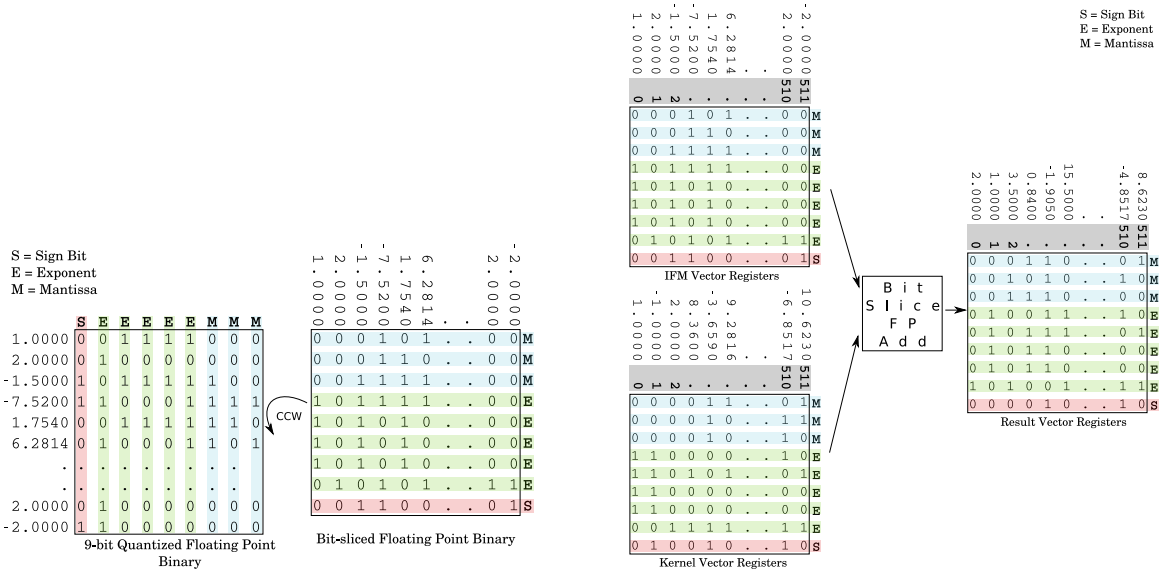
(c) **AVX512 3-Input LUT Bitwise Operations.**

Figure 4-2: Full Adders (Implemented in Intel's AVX2 and AVX512 and Arm's Neon Intrinsic Bitwise Operations.)

should be used. We create all 256 equivalent cells in the Liberty cell library when targeting AVX512 devices. Table 4.1 lists the bitwise operations supported in the cell libraries for each architecture. The AVX512 column shows a truncated example subset of the available 256 bitwise logic instructions [Intel 2020] in both hardware and software cell libraries.

To demonstrate the capabilities of the cell libraries, we show an example of a full adder. Figure 4-2a shows a typical 5-gate full adder implemented with our AVX2 cell library.

The same full adder can be implemented in three Arm Neon bitwise logic instructions, Figure 4-2b, one of which is the SEL bitwise multiplexer instruction. Intel AVX512 intrinsics can implement the full adder in two 3-input bitwise ternary instructions, Figure 4-2c. While the inputs and outputs of the hardware gate level circuits are single bits, these inputs and outputs are parallelised by the bit-width of



(a) **Many 9-bit FP Values Transformed to 9-bit bitsliced FP Data Layout.**

(b) **512 9-bit Bitslice FP Add Operation Using AVX512 Registers.**

Figure 4-3: Bit-sliced Parallel FP Transformation and Bit-sliced FP Add Operation.

the SIMD vector registers. The bitwise instructions, converted to bitwise software operations, produce extensive parallel scaling of the arithmetic.

4.3.2 HOBFLOPS Bitslice Parallel Operations

HOBFLOPS exploits bitslice parallel operations to represent FP numbers in a bitwise manner that are processed in parallel. For example, many 9-bit values are transformed to bitslice parallel representations, see Figure 4-3a. A simple example of how nine registers of 512-bit bitslice parallel data are applied to a 512-bit wide bitslice parallel FP adder is shown in Figure 4-3b. Each adder instruction has a throughput of between a 1/3 and 1 clock cycle [Intel 2020; Arm 2019]. In this example, the adder's propagation delay is related to the number of instruction-level parallelism and associated load/store commands. The number of gates and thus SIMD vector instructions in the HOBFLOPS adder or multiplier is dependent on the required HOBFLOPS precision. See Table 4.3 for examples of HOBFLOPS MAC

Table 4.2: Bit Width Comparisons of Existing Custom FP.

Sign	Exponent	Mantissa	Type, Availability, Performance
1	4	3	IEEE-FP8 (Slow in S/W) [IEEE 2019]
1	5	2	MS-FP8 (Fast in H/W) [Chung et al. 2018; Fowers et al. 2018]
1	5	3	MS-FP9 (Fast in H/W) [Chung et al. 2018; Fowers et al. 2018]
1	5	10	SoftFP16 (Slow in S/W) [Hauser 1999]
1	8	23	SoftFP32 (Slow in S/W) [Hauser 1999]
1	Arbitrary	Arbitrary	Flexfloat (Slow in S/W) [Tagliavini et al. 2018]
1	Arbitrary	Arbitrary	HOBFLOPS (Fast in S/W) **(Ours)**

Table 4.3: HOBFLOPS MAC Standard and Extended Range and Precision Types

hobflops(IEEE)*XX*	**Inputs Bit Width**		**Outputs Bit Width**		**hobflops(IEEE)*XXe***	**Inputs Bit Width**		**Outputs Bit Width**	
	Expo	**Mant**	**Expo**	**Mant**		**Expo**	**Mant**	**Expo**	**Mant**
HOBFLOPSIEEE8	4	3	4	4	HOBFLOPSIEEE8e	4	3	4	7
HOBFLOPS8	5	2	5	3	HOBFLOPS8e	5	2	5	5
HOBFLOPS9	5	3	5	4	HOBFLOPS9e	5	3	5	7
... (truncated)	...	...	...	...	***... (truncated)***	...	...	...	...
HOBFLOPS16	5	10	5	11	HOBFLOPS16e	5	10	5	21

precision, and section 4.4 for associated HOBFLOPS MAC SIMD vector instruction counts and performance.

4.3.3 HOBFLOPS Design Flow

Taking inspiration from Microsoft's MS-FP8 / MS-FP9 [Chung et al. 2018; Fowers et al. 2018] and Minifloat 8-bit that follows the IEEE-754 2019 specification, we create single- and extended-precision HOBFLOPS adders and multipliers. For example, we create the single-precision HOBFLOPS8 multiplier to take two 5-bit exponent, 2-bit mantissa, 1-bit sign inputs and produce a single 5-bit exponent and 3-bit mantissa and 1-bit sign output. We also create extended-precision HOBFLOPS8e multiplier to take two 5-bit exponent, 2-bit mantissa, and 1-bit sign to produce a single 5-bit exponent, 5-bit extended mantissa and 1-bit sign output.

When using HOBFLOPS, any quantisation method may be employed, such as those investigated by Gong *et al.*, [Gong et al. 2014]. These quantised values are stored and computed during inference mode. Therefore we set FloPoCo to the required range and precision. For comparison, Table 4.2 shows the range and precision of IEEE-FP8 of MS-FP8 and MS-FP9, respectively, available in simulated software and FPGA hardware. These FP solutions only support specific ranges and do not allow for arbitrary exponent and mantissa values.

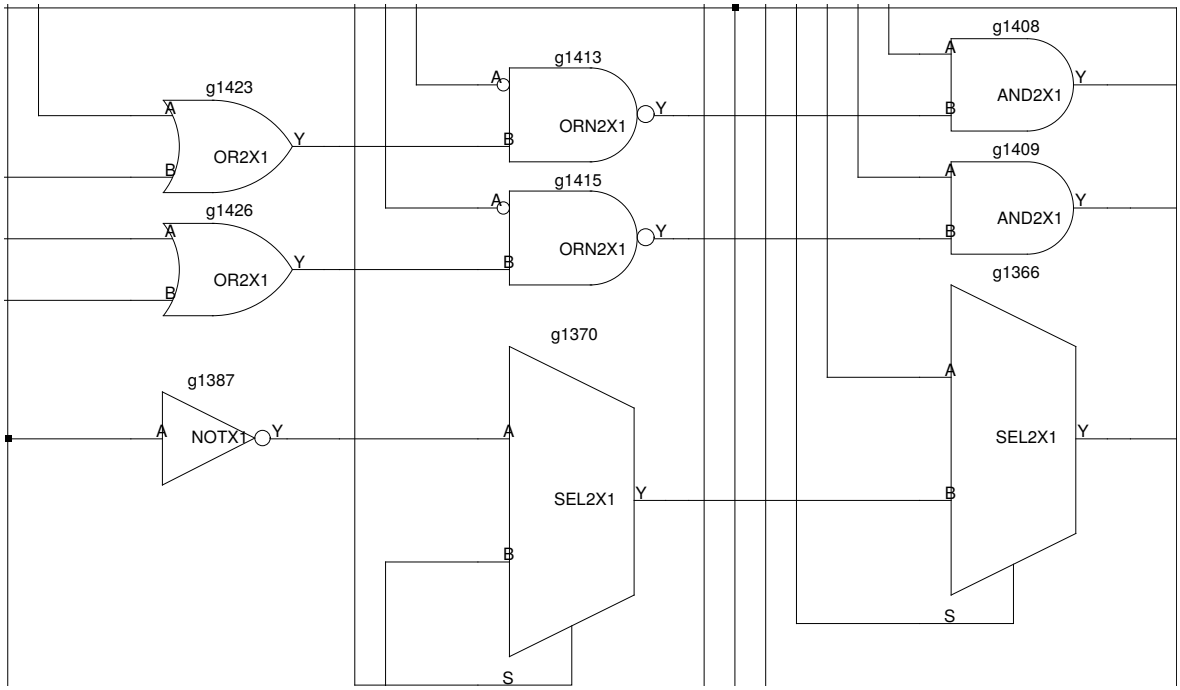


Figure 4-4: **Zoomed Snippet of An Example HOBFLOPS8 Multiplier Netlist Using Arm Neon Cells.**

Details of the evaluated HOBFLOPS are in Table 4.3, although, other arbitrary combinations of mantissa and exponent bit-widths are supported in the flow (see Figure 4-1).

FloPoCo [De Dinechin et al. 2009], generates VHDL descriptions of FP adders and multipliers of varying exponent and mantissa bit-widths for standard and extended precision HOBFLOPS types, see Table 4.3. As a performance baseline, we produce the 16- and 16e-bit IEEE-754 FP versions of the multiplier and adder, for comparison against Flexfloat and SoftFP. FloPoCo automatically produces the corresponding VHDL test benches and test vectors required to test the generated cores. FloPoCo has a slightly different way of encoding the FP numbers when compared to the IEEE-754 2019 specification and does not support subnormal numbers. A FloPoCo FP number [De Dinechin et al. 2009] is a bit-vector consisting of the following 4 fields: 2-bit exception field (*01* for normal numbers); a sign bit; an exponent field *wE* bits wide; a mantissa (fractional) field *wF* bits wide. The significand has an implicit leading *1*, so the fraction field *ff...ff* represents the significand *1.ff...ff*.

We configure FloPoCo to generate combinatorial plain RTL VHDL cores with a 1MHz frequency, no pipelining, and no use of hard-macro FPGA multipliers or adders, Listing 4.1. These settings ensure that FloPoCo generates reduced area

rather than reduced latency multipliers and adders. We simulate the FP multipliers and adders in a VHDL simulator with the corresponding FloPoCo generated test bench to confirm that the quantised functionality is equivalent to IEEE-754 FP multiplier and adder.

We create Synopsys Liberty standard cell libraries to support the target processor architecture, e.g., Listing 4.6. Cadence Genus (version 16.22-s033_1) the industry-standard ASIC synthesis tool, synthesises the adder and multiplier VHDL cores with our standard cell libraries and configuration and small gate area optimisation script, Listing 4.3, into a Verilog netlist of the logic gates. See Figure 4-4 for an example of the HOBFLOPS8 multiplier logic produced by FloPoCo when synthesised with our Arm Neon cell Library. Note how Genus has synthesised the design to include the 3-input SEL gate (multiplexer) supported by Arm Neon.

HOBFLOPS designs are combinatorial, so synthesis timing constraints are unnecessary. In the standard cell libraries Liberty file, we assign a value of 1.0 to *cell area* and *cell leakage power* of the cells. We configure the cell capacitance and timing values to zero. These values ensure the synthesizer assigns equal optimisation priority to all gates and produces a netlist with the least number of logic gates rather than creating a netlist optimised for hardware timing propagation.

We further optimise the netlist using the open-source Yosys ASIC synthesizer [Wolf et al. 2013] and ABC optimiser [Brayton and Mishchenko 2010]. We use ABC's *strash* command to transform the current network into an AND-inverter graph (AIG) by one-level structural hashing. We then use the *refactor* function to iteratively collapse and refactor the levels of logic and area of the netlist. We configure Yosys to produce a topologically sorted Verilog netlist of gates, Listing 4.4. The topological sorting is required as Cadence Genus writes the netlist file in an output port to input port order, whereas the C/C++ compiler requires the converted netlist to have input to output ordering. We formally verify the topologically sorted netlist against the original netlist with Yosys satisfiability (SAT)-solver. These netlists are re-simulated with the test bench used to simulate the FloPoCo generated VHDL designs and compared for correlation.

Our domain-specific source-to-source generator, Listing 4.5, translates the Verilog adder and multiplier netlists to Intel AVX2, AVX512, or Arm Neon bitwise oper-

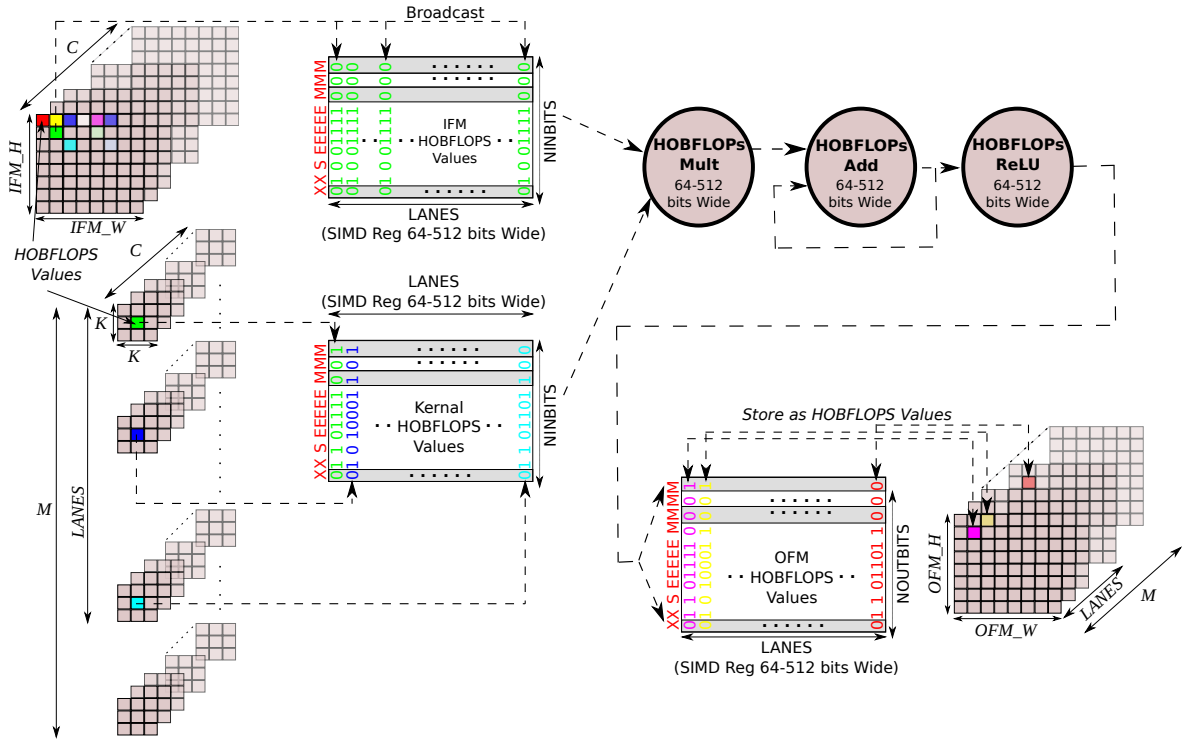


Figure 4-5: **HOBFLOPS CNN Convolution (IFM and Kernel data pre-transformed to HOBFLOPS, output feature map (OFM) remains in HOBFLOPS layout for use in the next layer. IFM values are broadcast across the kernel values. Arm Neon SIMD LANES are up to 128-bit wide and Intel SIMD LANES are up to 512-bit wide.)**

ALGORITHM 1: HOBFLOPS Code for a Simple AVX2 2-bit Binary Full Adder (unrolled by Synthesis) - 256 wide 2-bit adders in 12 Bitwise Operations.

Input: *x[2]* of SIMD width
Input: *y[2]* of SIMD width
Input: *cin* of SIMD width
Output: *HOBFLOPS* register *sum[2]* of SIMD width
Output: *HOBFLOPS* register *cout* of SIMD width
XOR2X1(x[1], y[1], n_5); *// Wide XOR Operation*
OR2X1(x[1], y[1], n_2); *// Wide OR Operation*
OR2X1(cin, x[0], n_1);
AND2X1(x[1], y[1], n_0); *// Wide AND Operation*
AND2X1(cin, x[0], n_3);
AND2X1(y[0], n_1, n_6);
OR2X1(n_3, n_6, n_8);
AND2X1(n_2, n_8, n_9);
OR2X1(n_0, n_9, cout);
XOR2X1(n_5, n_8, sum[1]);
XOR2X1(x[0], y[0], n_4);
XOR2X1(cin, n_4, sum[0]);

ators with the correct types and pointers. Algorithm 1 shows the HOBFLOPS code for a simple 2-bit binary adder with carry, generated from 12 bitwise operations, referenced from the cell library of Listing 4.6. A single input bit of the hardware multiplier becomes the corresponding architecture variable type *e.g.*, uint64 for a 64-bit processor, an __mm256i type for an AVX2 processor, an __mm512i type for an AVX512 processor, uint32x4_t type for a Neon processor (see also Figure 4-2). Algo-

ALGORITHM 2: HOBFLOPS Code for a Simple AVX512 2-bit Binary Full Adder - 512 wide 2-bit adders in 4 Bitwise Operations.

Input: *x[2]* of SIMD width
Input: *y[2]* of SIMD width
Input: *cin* of SIMD width
Output: *HOBFLOPS* register *sum[2]* of SIMD width
Output: *HOBFLOPS* register *cout* of SIMD width
LUT232X1(cin, y[0], x[0], n_1); // $(B\&C)|A\&(B \wedge C)$
LUT232X1(x[1], n_1, y[1], cout);
LUT150X1(y[1], x[1], n_1, sum[1]); // $A \wedge B \wedge C$
LUT150X1(y[0], x[0], cin, sum[0]);

rithm 2 demonstrates the gate-count and thus the SIMD operational efficiency of the AVX512 implementation of the same simple 2-bit adder, generated from 4 bitwise operations.

A HOBFLOPS8 multiplier targeted at the *e.g.*, AVX2 processor is generated in 80 bitwise operations, and a HOBFLOPS8 adder is generated in 319 bitwise operations. When targeted at the AVX512 processor, the HOBFLOPS8 multiplier is generated in 53 bitwise operations, and the HOBFLOPS8 adder is generated in 204 bitwise operations.

4.3.4 CNN Convolution with HOBFLOPS

We present a method for CNN convolution with HOBFLOPS arithmetic. We implement HOBFLOPS MACs in a CNN convolution layer, where up to 90% of the computation time is spent in a CNN [Farabet et al. 2010]. We compare HOBFLOPS MAC performance to IEEE FP MAC and to Flexfloat [Tagliavini et al. 2018] and SoftFP [Hauser 1999].

Figure 4-5 shows HOBFLOPS IFM and kernel values convolved and stored in the OFM. To reduce cache misses, we tile the IFM $H \times W \times C$ dimensions, which for *Conv dw / s2* of MobileNets CNN is $14 \times 14 \times 512 = 100,352$ elements of HOBFLOPS IFM values. We tile the M kernel values by $LANES \times NINBITS$, where $LANES$ corresponds to the target architecture registers bit-width, *e.g.*, 512-lanes corresponds to AVX512 512-bit wide register. The $LANES \times NINBITS$ tiles of binary values are transformed to $NINBITS$ of *SIMD width* values, where *SIMD width* correspond to the target architecture register width, *e.g.*, uint64 type for a 64-bit processor architecture, __mm512i type for AVX512 processor.

We broadcast the IFM channel tile of $NINBITS$ across the corresponding channel of all the kernels tiles of $NINBITS$ to convolve image and kernel values using HOBFLOPS multipliers, adders and ReLU activation function. The resultant convo-

lution *SIMD width* values of $NOUTBITS$ wide are stored in corresponding location tiles in the OFM. The HOBFLOPS IFM and kernel layout for single-precision, as defined by FloPoCo is outlined in Equation 1.

$$NINBITS=EXC+SIGN+EXPO\_IN+MANT\_IN \tag{1}$$

The HOBFLOPS OFM layout for single-precision is shown in Equation 2.

$$NOUTBITS=EXC+SIGN+EXPO\_IN+MANT\_IN+1 \tag{2}$$

and for extended-precision, see Equation 3.

$$NOUTBITS=EXC+SIGN+EXPO\_IN+(2 \times MANT\_IN)+1 \tag{3}$$

For example, HOBFLOPS9, as can be seen in Table 4.3, the input layout $NINBITS$ has 2-bit exception EXC, 1-bit sign $SIGN$, 5-bit exponent $EXPO\_IN$, 3-bit mantissa $MANT\_IN$, which added comes to 11-bits. The single-precision $NOUTBITS$ would be 12-bits ($NINBITS + 1$). The extended-precision $NOUTBITS$ would be 15-bits.

If HOBFLOPS is implemented in a multi-layer CNN, the data between each layer could remain in HOBFLOPS format until the last convolution layer. The OFM at the last convolutional layer could be transformed from HOBFLOPS values to floats resulting in the transformation overhead only occurring at the first and last convolutional layers of the CNN model. An additional pooling layer could be developed in the HOBFLOPS format, for the interface between the last convolutional layer and the fully connected layer, not done in this work.

We repeat for MACs up to HOBFLOPS16e on each processor architecture up to the 512-bit wide AVX512 registers.

4.4 Evaluation

We implement each of the 8- to 16e-bit HOBFLOPS multipliers and adders in a convolution layer of the MobileNets CNN [Howard et al. 2017]. We use layer *Conv dw / s2* of MobileNets as it has a high number of channels C and kernels M, perfect

for demonstrations of high-dimensional parallelised MAC operations. We compare the HOBFLOPS16 multipliers and adders round-to-nearest-ties-to-even and round-towards-zero modes performance to SoftFP16 rounding near_even and round min modes [Hauser 1999]. This 16-bit FP comparison acts as a baseline as Soft FP8 is not supported by Berkeley's emulation tool.

We target 32-bit to 512-bit registers for AVX2 and AVX512 processors and target 32- and 128-bit registers for the Cortex-A15 processor. We implement 32- to 512-lanes of HOBFLOPS multipliers and adders and capture each of the AVX2 32-, 64-, 128- and 256-bit, AVX512 32-, 64-, 128-, 256 and 512-bit, and Cortex-A16 32-, 64- and 128-bit results.

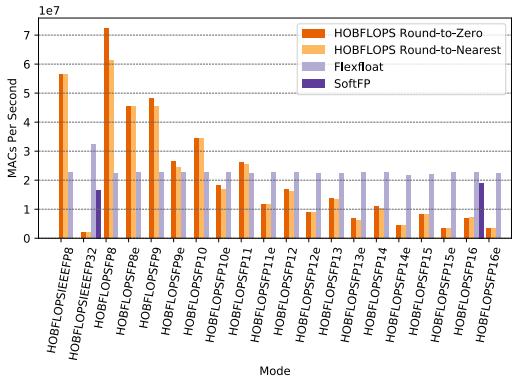
Three machine types are used to test the HOBFLOPS MAC:

- Arm Cortex-A15 Neon embedded development kit, containing ARMv7 rev 3 (v7l) CPU at 2GHz and 2GB RAM;
- Intel Core-i7 PC, containing Intel Core-i7 8700K CPU at 3.7GHz and 32GB RAM;
- Intel Xeon Gold server PC, containing Intel Xeon Gold 5120 at 2.2GHz and 256GB RAM.

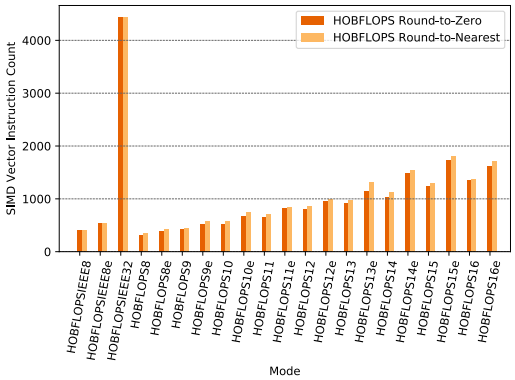
Our cell libraries model-specific cells, see Table 4.1. We omit the bit clear (BIC) of the Arm Neon in our cell library. Inclusion of bit clear (BIC) prevents the synthesis tool, Cadence Genus, from optimising the netlist with SEL (multiplexer) units, leading to a less area efficient netlist.

To further decrease area and increase performance, we produce round-towards-zero versions of the HOBFLOPS8–HOBFLOPS16e adders as the rounding can be dealt with at the end of the layer in the activation function, assuming the non-rounded part of the FP value is retained through to the end of the layer.

The MACs per second of an average of 1000 iterations of a HOBFLOPS adders and multipliers are captured and compared. We use the GNU G++ compiler to optimise the code for the underlying target microprocessor architecture and numbers of registers. We compile the HOBFLOPS CNN code (see Figure 4-5) to include our HOBFLOPS adders and multipliers, and our cell library (*e.g.*, Listing 4.6 for AVX2) with G++ (version 8.2.1 20181127 on Arm, version 9.2.0 on Intel AVX2 and version 6.3.0 20170516 on Intel AVX512 machines). We target C++ version 17 and



(a) **Throughput of Arm Neon, HOBFLOPSIEEE8-32, and HOBFLOPS8-16e MACs/s Compared to Flexfloat and SoftFP MACs/s - higher throughput is better.**



(b) **Arm Neon SIMD Bitwise Vector Instruction Count HOBFLOPSIEEE8-32, and HOBFLOPS8-16e MACs - lower gate count is better.**

Figure 4-6: Throughput and SIMD-Bitwise-Vector-Instruction Count of Arm Neon, HOBFLOPSIEEE8-32, and HOBFLOPS8-16e MACs in Convolution of Two Rounding Methods.

using *-march=native, -mtune=native, -fPIC, -O3* compiler switches with $-msse$ for SSE devices and *-mavx2* for AVX2 devices. When targeting an Intel AVX512 architecture we use the the *-march=skylake-avx512, -mtune=skylake-avx512, -mavx512f, -fPIC, -O3* switches. When targeting an Arm Neon device we use *-march=native, -mtune=native, -fPIC, -O3, -mfpu=neon* to exploit the use of Neon registers.

After the G++ compilation, we inspect the assembler object dump. Within the multiplier and adder units, we find an almost one-to-one correlation of logic bitwise operations in the assembler related to the gates modelled in the cell libraries, with additional loads/stores where the compiler has seen fit to implement.

4.4.1 Arm Cortex-A15 Performance

We configure an Arm Cortex-A15 Development kit with 2GB RAM, ARCH Linux version 4.14.107-1-ARCH installed, and fix the processor frequency at 2GHz. We run tests for 32-, 64- and 128-lanes and capture performance. We use *taskset* to lock the process to a core of the machine for measurement consistency.

Figure 4-6a shows 128-lane performance for all arbitrary-precision HOBFLOPS FP between 8- and 16e-bits, IEEE 8- and 32-bit equivalents, Flexfloat, and SoftFP versions. HOBFLOPS16 round-to-nearest-ties-to-even achieves approximately half

the performance of Flexfloat and SoftFP 16-bit rounding near_even mode on Arm Neon. However, HOBFLOPS offers arbitrary precision mantissa and exponent FP between 8- and 16-bits, outperforming Flexfloat and SoftFP between HOBFLOPS8 and HOBFLOPS12 bits.

Similarly, HOBFLOPS16 round-towards-zero version shown in Figure 4-6b demonstrates a slight improvement in performance compared to Flexfloat and SoftFP rounding min mode. Figure 4-6b also shows HOBFLOPS round-towards-zero versions have an increased performance when compared to HOBFLOPS round-to-nearest-ties-to-even.

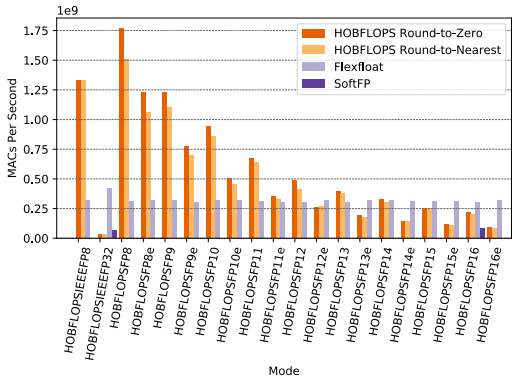
HOBFLOPS appears to exhibit a fluctuation around HOBFLOPS8e and HOBFLOPS9 in Figure 4-6b. While there is 1-bit more in the input mantissa of HOBFLOPS9 compared to HOBFLOPS8e, which leads to HOBFLOPS9 containing larger adder/accumulators, Figure 4-6b shows the round-towards-zero HOBFLOPS9 functionality almost counter-intuitively exhibiting slightly greater throughput than HOBFLOPS8e. The greater throughput of the round-towards-zero HOBFLOPS9 is due to the lack of rounding adder, reduced gate area and latency.

The low bit-width and thus reduced hardware synthesis gate count or area as seen in Figure 4-6 would benefit memory storage and bandwidth within the embedded system. This allows for reduced energy consumption, however, energy consumption is not measured here.

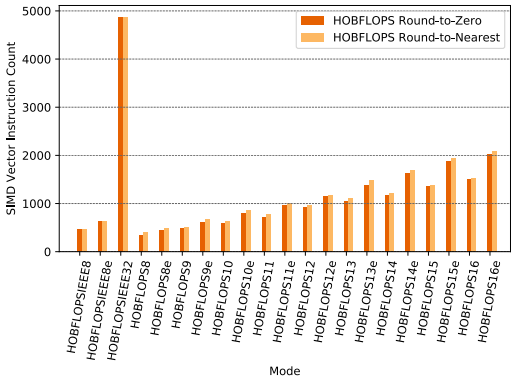
4.4.2 Intel AVX2 Performance

We configure an Intel Core i7-8700K desktop machine with 32GB RAM, and ARCH Linux 5.3.4-arch1-1 installed. For consistency of performance measurements of various HOBFLOPS configurations, within the BIOS we disable:

- Intel's SpeedStep (*i.e.*, prevent the CPU performance from ramping up and down);
- Multi-threading (*i.e.*, do not split the program into separate threads);
- TurboBoost (*i.e.*, keep all processor cores running at the same frequency);
- Hyperthreading Control (*i.e.*, keep one program on one processor core);
- C-States control (*i.e.*, prevent power saving from ramping down the core clock frequency).



(a) **Throughput of Intel AVX2, HOBFLOPSIEEE8-32, and HOBFLOPS8-16e MACs/s Compared to Flexfloat and SoftFP MACs/s - higher throughput is better.**

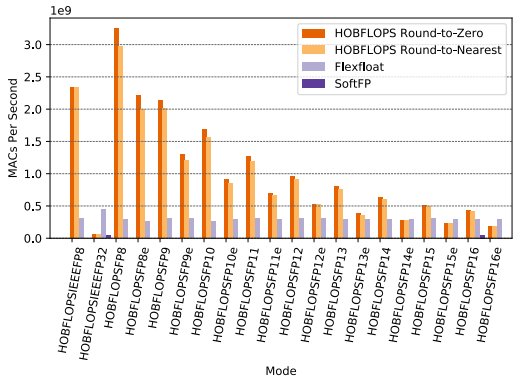


(b) **Intel AVX2 SIMD Bitwise Vector Instruction Count of HOBFLOPSIEEE8-32, and HOBFLOPS8-16e MACs - lower gate count is better.**

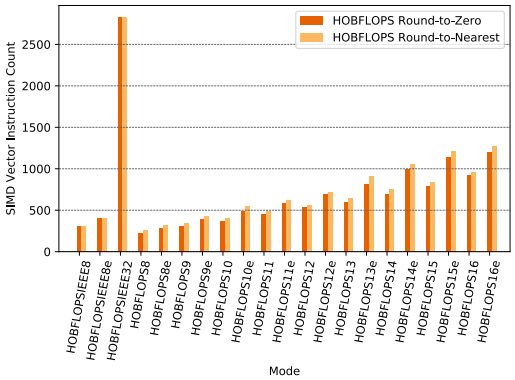
Figure 4-7: Throughput and SIMD-Bitwise-Vector-Instruction Count of Intel AVX2, HOBFLOPSIEEE8-32, and HOBFLOPS8-16e MACs in Convolution of Two Rounding Methods.

We alter GRUB's configuration so *intel_pstate* (*i.e.*, lock the processor core clock frequency) and *intel_cstate* are disabled on both *GRUB_CMDLINE_LINUX* and *GRUB_CMDLINE_LINUX_DEFAULT*. This BIOS and Linux Kernel configuration ensures the processor frequency is fixed at 4.6GHz, no power-saving, and each HOBFLOPS instance running at full performance on a single thread and single CPU core. When executing the compiled code, *taskset* is used to lock the process to a single core of the CPU. These configurations allow a reproducible comparison of timing performance of each HOBFLOPS configuration against SoftFP16.

We run tests for 32-, 64-, 128- and 256-lanes and capture performance. Figure 4-7a shows 256-lane results for all arbitrary-precision HOBFLOPS FP between 8- and 16e-bits, IEEE 8- and 32-bit equivalents, Flexfloat, and SoftFP versions. HOBFLOPS16 performs over $2.5\times$ higher MACs/second when compared to Berkeley's SoftFP16 *MulAdd* rounding near_even mode. The round-towards-zero version of HOBFLOPS16 performs at around $2.7\times$ higher MACs/second when compared to Berkeley's SoftFP16 *MulAdd* rounding min mode. In fact, HOBFLOPS outperforms SoftFP for all versions of between HOBFLOPS8 and HOBFLOPS16 for both round-to-nearest-ties-to-even and round-towards-zero rounding modes. HOBFLOPS outperforms Flexfloat up to HOBFLOPS12e for both rounding modes.



(a) **Throughput of Intel AVX512, HOBFLOPSIEEE8-32, and HOBFLOPS8-16e MACs/s Compared to Flexfloat and SoftFP MACs/s - higher throughput is better.**



(b) **Intel AVX512 SIMD Bitwise Vector Instruction Count of HOBFLOPSIEEE8-32, and HOBFLOPS8-16e MACs - lower gate count is better.**

Figure 4-8: Throughput and SIMD-Bitwise-Vector-Instruction Count of Intel AVX512 HOBFLOPSIEEE8-32 and HOBFLOPS8-16e MACs in Convolution of Two Rounding Methods.

HOBFLOPS performance gain is due to reduced synthesis area of the HOBFLOPS units as seen in Figure 4-7b. Again, this reduction in area, also seen for round-towards-zero, is key to reduced SIMD bitwise operations being created for the HOBFLOPS MACs and therefore reduced latency through the software HOBFLOPS MACs.

4.4.3 Intel AVX512 Performance

We configure an Intel Xeon Gold 5120 server with 256GB RAM, and Debian Linux 4.9.189-3+deb9u2. This shared server-grade machine BIOS or clock could not be changed as done for the AVX2-based machine. However, *taskset* is used to lock the process to a single CPU core.

We run tests for 32-, 64-, 128-, 256- and 512-lanes. Figure 4-8a captures 512-lane results for HOBFLOPS FP between 8- and 16e-bits, IEEE 8- and 32-bit equivalents and, Flexfloat, and SoftFP versions. HOBFLOPS16 performs with $8.2\times$ greater MACs throughput than SoftFP16 *MulAdd* rounding near_even mode. For the 512-lane round-towards-zero results, HOBFLOPS16 performs at $8.4\times$ the MACs throughput of SoftFP16 *MulAdd* rounding min mode. HOBFLOPS significantly outperforms Flexfloat for HOBFLOPS8 to HOBFLOPS16 for both round-to-nearest-ties-to-even

and round-towards-zero. HOBFLOPS9 performs at approximately 2 billion MACs/second, around 7× the performance of Flexfloat 9-bit.

HOBFLOPS performance is due to HOBFLOPS lower hardware synthesis area. When the netlists are converted to software bitwise operations, HOBFLOPS translates to fewer SIMD 3-input ternary logic LUT instructions the MACs. Figure 4-8b shows HOBFLOPS16 area on the AVX512 platform is 38% smaller than HOBFLOPS16 area on AVX2. A further slight performance boost is seen for round-towards-zero.

4.5 Related Work

As discussed in section 2.1 starting on page 9, CNNs require a great deal of computation, so, many researchers focus on proposing methods of optimising FP arithmetic within the CNNs either at the register level or the bit-precision level. Traditionally, researchers propose SIMD register and data path optimisations from within the compiler. An early example from Fisher *et al.*, [1998] is the SIMD within a register (SWAR) model which treats a wide data path within a processor as multiple, thin SIMD parallel data paths. This model accelerates arithmetic, allowing wide registers to be partitioned for smaller operations, such as subdividing 32-bit integer adders into eight 8-bit adders, assuming one adder's carry to the next adder is suppressed. The work of Fisher *et al.*, is an early work of efficient register packing which inspires the bitslice packing of our work.

Researchers investigate different bit precision and representations of the FP number base. As outlined in section 2.1, Google's TPU ASIC implements bfloat16 [Google 2019], preserving dynamic range but reducing the precision. Bfloat16 only reduces the bit-width of the mantissa and does not investigate different exponent or mantissa bit-widths. Our work addresses these arbitrary exponent and mantissa bit-widths.

Nvidia has proposed the new TP32 number representation [NVidia 2020], which is a sign bit, 8-bit exponent and 10-bit mantissa. This 19-bit floating-point format is an input format which truncates the IEEE FP32 mantissa to 10-bits. TP32 still produces 32-bit floating-point results to move and store in the GPU. Similar to Google,

Nvidia does not investigate other bit-widths of FP range and precision, something our work does.

As we show in section 2.1, typically 32-bit or 64-bit precision FP arithmetic is used for inference or training of a CNN. Johnson [2018] shows that changes in range and precision that lead to increased performance, typically lead to overall reduced energy consumption.

Note there are converter tools, such as Verilator, that converts hardware description language (HDL) code like Verilog to C/C++. Verilator does not merely convert Verilog netlists to C/C++ as Verilator's focus is on fast, optimised, threaded compiled C++ model of Verilog for simulation purposes. Verilator does not produce a circuit in a format convertible to bitwise instructions, something our custom code-generator does.

4.6 Conclusion

Arbitrary precision floating-point computation is largely unavailable in CPUs or FPUs. FP simulation latency is often too high for computationally intensive systems such as CNNs. FP simulation has little support for 8-bit or arbitrary precision.

We propose HOBFLOPS, that generates fast custom-precision bitslice-parallel software FP arithmetic. We optimise HOBFLOPS using a hardware design flow, cell libraries and custom code-generator. We generate gate-count in hardware and thus low SIMD operationally efficient software-emulated FP operators with an arbitrary precision mantissa and exponent. We pack the generated FP operators efficiently into the target processors SIMD vector registers. HOBFLOPS offers FP with custom range and precision, useful for FP CNN acceleration where memory storage and bandwidth are limited. Low-precision FP arithmetic in the MAC offers increased classification accuracy compared with INT8 or other number representation counterparts. HOBFLOPS saves machine operations for a comparable convolution compared to its counterparts.

We experiment with large numbers of channels and kernels in CNN convolution. We compare the MAC performance in CNN convolution on Arm and Intel processors against Flexfloat, and Berkeley's SoftFP. HOBFLOPS outperforms Flexfloat by over $10\times$, $5\times$, and $3\times$ on Intel AVX512, AVX2 and Arm Neon respectively. HOB-

FLOPS offers arbitrary-precision FP with custom range and precision, *e.g.*, HOBFLOPS9, which outperforms Flexfloat 9-bit by over $7\times$, $3\times$, and $2\times$ on Intel AVX512, AVX2 and Arm Neon respectively.

The performance gains are due to the optimised hardware synthesis area of the MACs, translating to fewer bitwise operations. Additionally, the bitslice parallelism of the very wide vectorisation of the MACs of CNN convolution contributes to the performance boost. While we show results for 8- and 16-bit with a fixed exponent, HOBFLOPS supports the emulation of any required precision of FP arithmetic at any bit-width of mantissa or exponent, *e.g.*, FP9 containing a 1-bit sign, 5-bit exponent and 3-bit mantissa, or FP11 containing a 1-bit sign, 5-bit exponent and 5-bit mantissa, not efficiently supported with other software FP emulation.

Other processors, *e.g.*, RiscV or Arm SVE (when available) can be supported by producing a standard cell library of the processors bitwise vector extensions. HOBFLOPS allows researchers to prototype different levels of custom FP precision in the arithmetic of software CNN accelerators. Furthermore, HOBFLOPS fast custom-precision FP CNNs may be valuable in cases where memory bandwidth is limited. The application of HOBFLOPS could therefore be extended to areas outside of CNN acceleration, such as GPGPU acceleration, however, this has not been investigated.

5 Conclusion

5.1 Contributions and Discoveries

THIS thesis investigates opportunities for improving the energy efficiency and gate-level area of the CNN convolution layer MAC arithmetic, when implemented in embedded hardware such as an ASIC, FPGA. We also investigate opportunities to improve the execution time and floating-point precision of the CNN convolution layer arithmetic in software on embedded and high-end CPUs.

The convolution layer consists of millions of MACs, the arithmetic of which can be in fixed-point, integer or floating-point format. The CNN can operate in training mode or inference mode. During inference mode, the convolution layer occupies up to 90% of the computation time of the CNN, convolving the input feature map (IFM) with the kernel weight data. The storage and movement of weight data and computation of the convolution layer are often beyond the energy, storage and compute bounds of edge devices. We investigate two areas for optimising the MAC arithmetic of the convolution layer while maintaining FP accuracy in inference mode, for CNN accelerator implementation in edge devices:

- For successful CNN inference, large quantities of weight data have to be moved from RAM to the layers of the CNN. Researchers optimise the energy consumption of data transfers from RAM with a data compression method of weight-sharing [Han et al. 2016a]. However, weight-sharing does not approach optimising the arithmetic of the weight data with the IFM within the convolution operation. Our PASM work proposes the optimisation of the computational arithmetic of a weight-shared CNN. PASM focuses on optimising

the power and area of ASIC and FPGA implementations of the convolution layer MAC arithmetic. We demonstrate how we build on the weight-sharing scheme to reduce the numbers of multiply operations required in the MAC of the convolution layer. We show that our PASM accelerator consumes up to 66% fewer logic gates in ASIC and 99% less energy consumption than the equivalent weight-shared MAC, with up to a 12% increase in latency. We originally planned to decrease ASIC area and increase energy efficiency. However, the discovery of reduced DSP utilisation and therefore energy in FPGA suggests that PASM could be quickly and cheaply implemented in accelerators before engineering silicon of the ASIC accelerator implementation is available;

- Due to the computational, bandwidth, and energy restrictions of 32-bit FP in embedded systems, researchers have trended towards reduced precision fixed-point arithmetic in CNNs, often at the expense classification accuracy. Bitsliced FP operations are more efficient and faster than FP operations and exhibit more range and precision than equivalent bit-width fixed-point arithmetic [Xu and Gregg 2017]. Our HOBFLOPS work proposes a method of producing low-precision floating-point arithmetic in embedded and high-end server software for CNN acceleration. HOBFLOPS is suitable for performance and reducing memory transfers with increased FP precision. HOBFLOPS is a software algorithm and code generation flow for implementing CNN convolution with the very wide vectors that arise in bitslice parallel vector arithmetic. We experiment with large numbers of channels and kernels in CNN convolution and show that HOBFLOPS outperforms Flexfloat by over $10\times$, $5\times$, and $3\times$ on Intel AVX512, AVX2 and Arm Neon, respectively. HOBFLOPS offers arbitrary-precision FP with custom range and precision, *e.g.*, HOBFLOPS9, which outperforms Flexfloat 9-bit by over $7\times$, $3\times$, and $2\times$ on Intel AVX512, AVX2 and Arm Neon respectively. This indicates that HOBFLOPS is very efficient for low-precision FP. We originally set out to use HOBFLOPS on embedded devices. However, we discovered that HOBFLOPS is also very efficient when implemented on high-end 512-bit processors, potentially saving energy in edge embedded devices and data centre applications.

5.2 Future Directions

We show that the PASM accelerator reduction in multipliers significantly reduces the hardware ASIC and FPGA area and energy consumption. While these PASM results are impressive, we measure efficiency benefits using the conservative post-synthesis Cadence design tools timing, power and area reports. We would see even better results if the energy measurements of PASM were measured when implemented in fully placed and routed ASIC silicon. Measurements of performance within the full hardware accelerator [Dinelli et al. 2020] for area, latency and energy efficiencies of PASM would highlight the effectiveness of an ASIC PASM CNN accelerator.

We implement HOBFLOPS with fixed range (5-bit exponent) [Chung et al. 2018] and vary the precision (mantissa bit width). The change of mantissa yields impressive results as the multiplier exists in the mantissa. It would be interesting to run the experiments with additional different bit widths of exponent and changing mantissa bit width. We show a 4-bit exponent for the IEEE-754 8-bit implementation and increase to a 5-bit exponent for the remaining results, however, the mantissa bit-width also changes. Investigating the RTL from FloPoCo suggests that an increase in bit-width of the exponent linearly increases the gate-level area for both FP multiplier and FP adder. The increase in the gate-level area of the exponent would suggest a linear increase in latency but with an increase in order-of-magnitude of supported range for both HOBFLOPS adder and HOBFLOPS multiplier.

The HOBFLOPS proposal would allow for the scaling of different arbitrary bit-widths of mantissa and exponent in different layers of a CNN. One might experiment with *e.g.*, higher precision HOBFLOPS for early layers and scaling to low precision for later layers of the CNN.

Another interesting area to investigate might be to mix the Xilinx FPGA FP operator IP with HOBFLOPS. Xilinx FP IP only support 16-, 32- and 64-bit FP arithmetic. HOBFLOPS can offer lower custom precision FP arithmetic, unsupported by Xilinx IP.

As the number of bits in the HOBFLOPS mantissa and exponent increases, HOBFLOPS becomes expensive in gate-level area, which translates to longer execution time in the subsequent software HOBFLOPS. Approximate arithmetic circuits typi-

cally guarantee precision for a few low-order bits *e.g.*, precision scaling of input and intermediate operands, lossy compression, inexact adder and multiplier [Xu et al. 2015; Mittal 2016]. It would be interesting to investigate the application of approximate arithmetic circuits to produce the HOBFLOPS arithmetic. The potential lower gate-level area of using approximate arithmetic in HOBFLOPS may shorten the execution time of larger mantissa and exponent HOBFLOPS software arithmetic.

Both the PASM and HOBFLOPS optimisations would benefit greatly from being implemented together to accelerate a weight-shared CNN. Comparisons of implementing PASM and HOBFLOPS on various microprocessors, FPGA and ASIC with combinations of PASM bins and HOBFLOPS range and precision should show benefit. Additionally, PASM and HOBFLOPS accelerators could also be applied to accelerate the convolution layers of Fast Region Based CNNs (RCNNs) [Girshick 2015] and You Only Look Once (YOLO) [Redmon et al. 2016].

Processing in-memory is a style of computer architecture where hardware for some processing capacity is incorporated into memory. Ambit [Seshadri et al. 2017] is a bulk bitwise accelerator-in-memory that performs AND, OR and NOT bitwise operations in DRAM. The cell library of HOBFLOPS could be configured to support such bitwise operations to produce HOBFLOPS custom-precision bitwise FP operations that can be performed in memory. Furthermore, approximate computing techniques identified above, such as an inexact adder and multiplier, could be used to implement HOBFLOPS in-memory, offering a great benefit to this area of DRAM accelerator research.

5.3 Final Thoughts

AI and the subset of ML are changing the world daily in many ways. ML inference at the edge needs to be fully achieved at low levels of application latency and energy. Researchers are focusing on producing lower power, lower bandwidth and lower storage ML models to aid the move of CNN acceleration closer to the edge. In our contributions, we have reduced the number of multipliers in a weight-shared CNN significantly reducing area and energy consumption with PASM. We have investigated implementing bitslicing strategies in the MAC of a CNN and show a significant reduction in execution time for CNN in Arm Neon and Intel processors.

Our work is a small and important component in transforming the world through AI at the edge.

Appendices

A

Neural Networks: Background and History

THE HISTORY of Automata and robots, right up to today's artificial intelligence and machine learning systems has fascinated the minds of humans for millennia. The historical review will show common threads throughout the research with a special interest in energy consumption and optimisation techniques applied to machine learning.

A.1 Introduction

For millennia, designers have strived to enable automata (usually in the form of robots) and machine learning systems to automate mundane tasks, entertain and, more recently, make medical diagnosis. These designs have been applied to several areas of ML in differing ways. Some fields of application, discussed below, are:

- Automata and robotics, *e.g.*, [Dynamics 2019];
- Gameplay to beat human champions [Kasparov and Greengard 2018; Wang et al. 2016b];
- Medical diagnostics and devices for detection of skin cancers [Esteva et al. 2017], brain cancers [Jermyn et al. 2016], breast cancers [Wang et al. 2016a], and most recently the Coronavirus [Maghdid et al. 2020], and genetic diseases, autism and spinal muscular atrophy [Xiong et al. 2015; Zhou and Troyanskaya 2015; Alipanahi et al. 2015; Zeng et al. 2016];
- Speech *e.g.*, [Hinton et al. 2012], natural language processing (NLP) [Collobert et al. 2011] and on-the-fly speech translation [Deng et al. 2013];

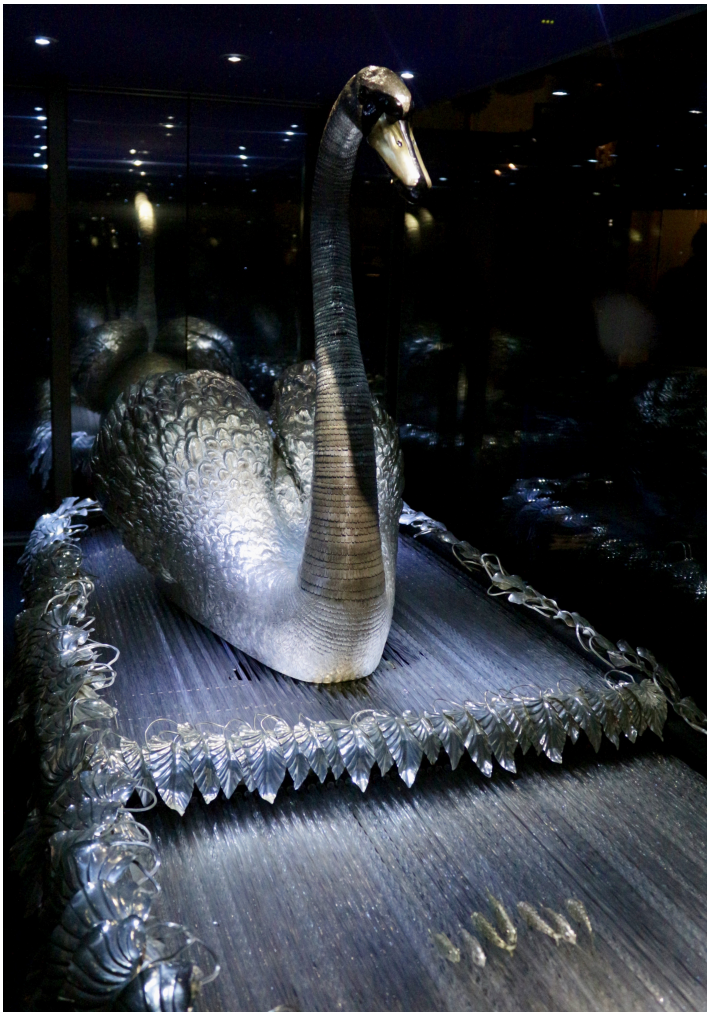


Figure A-1: **Silver Swan** (photo Stephen Curry).

- Image classification [Russakovsky et al. 2015], object localisation and detection [Girshick et al. 2014b], image segmentation [Long et al. 2015] and action recognition [Simonyan and Zisserman 2014a] in images and videos.

A.1.1 Automata and Robotics

Homer was the first to use the word Automata around 500 BC when describing the use of an automatic door [Bryant 1870a] and wheeled tripods [Bryant 1870b]. Hephaestus created Automata of walking tripods and maidens of gold for his workshop [Bryant 1870c]. Daedalus used quicksilver (now known as Mercury) to install

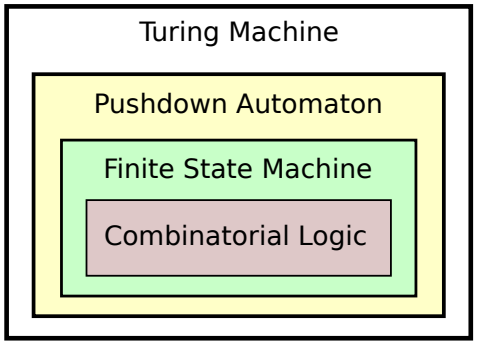


Figure A-2: **Classes of Automata Theory** [Wikipedia 2020].

voices in his moving statues [Kang 2002], and King Alcinous of the Phaeacians employed gold and silver watchdogs [Faraone 1987].

A self-operated automaton is designed to follow a programmed set of instructions. Humans have produced many iterations of Automata over the centuries, and few examples of early Automata remain. Some examples have stood the test of time. The Silver Swan by John Joseph Merlin, constructed in 1773 and housed in the Bowes Museum in England, is a clock-work life-size automaton of a swan sitting in a stream containing fish, Figure A-1. He made the stream from rotating glass rods, giving the illusion of flowing water. The swan's head bends down into the water to 'catch' one of the fish.

Another early example is Henri Maillardet's multi-tasking automaton that he designed and built in 1805. The automaton can draw four drawings and write three poems and has a life-size torso and head of a human. Maillardet's automaton is working and on display in the Franklin Institute Science Museum in Philadelphia, USA.

Automata spawned Automata theory or the study of abstract machines and computational problems. Figure A-2 shows some of the classes of Automata from the lowest level of abstraction, combinatorial logic that has a finite number of inputs and outputs, up to a Turing machine (more on the Turing Machine in subsection A.2.1).

Karel Capek, the Czech playwright, coined the word 'Robot' in his 1920 play Rossum's Universal Robot [Capek 2004]. The term robot comes from a combination of the Czech term for forced labour, 'robota', and the Slovak word for worker 'robotnik'. Robots, by today's definition, are machines designed to accomplish a task. However, the definition has gone further to define machines that use their

programming to make decisions. The sensor input guide the robot's *decisions*, the control systems make the *decisions*, and the actuators produce the desired output of the *decisions*. The sensors are designed to detect, for example, images and sounds. The actuators have to be fast and flexible enough to perform the task, and the control system has to make all the decisions to get the sensors and actuators working together, often in real-time and in the real world.

The WABOT-1 is considered the first full-scale *anthropomorphic robot* developed by researchers at Waseda University in Japan in 1973. It had arms and legs, could walk, pick things up with hands that contained tactile-sensors and could talk in Japanese. WABOT-1 was considered to have the mental capacity of a one-and-a-half-year-old child. It was slow, walking at a pace of 45 seconds per step and would only reply to specific questions with prerecorded statements. Later in 1980, the same group of researchers constructed WABOT-2, which was specialised to play the piano as piano playing was considered an activity that would require human-like dexterity and intelligence. The researchers concluded that it was easier to design robots to do one task at a time. This single task application is part of the reason today's robots are designed to do one job, such as vacuuming the floor or cleaning the dishes. These robots can be designed to be more efficient at these tasks than a general-purpose machine trying to perform the same task in similar power constraints. However, WABOT research allowed humans to interact with the robots verbally or physically and has led to the development of artificially intelligent systems.

George Devol's 'Unimation' robot [Devol 1961], was granted a patent in 1961. From this patent, Devol, in partnership with Joe Engelberger, developed Unimate. Unimate was a welding arm robot weighing in at a metric tonne. A magnetic drum containing the program told the robot arm to stack and weld the metal. The first industrial robot to be controlled by a computer was the IRB6 in 1974. The Swedish company, ABB developed the robot and gave it 16KB of RAM, made it programmable and gave it a display that could show four digits with LEDs [ABB 1974]. To perform more complex tasks than the tube polishing that the IRB6 performed would require the robot to have some form of a vision system. A good camera, lenses, and AI were needed to allow the camera to discern objects and interact with situations in real-time in the real world. By the late 1970s, research engineers had

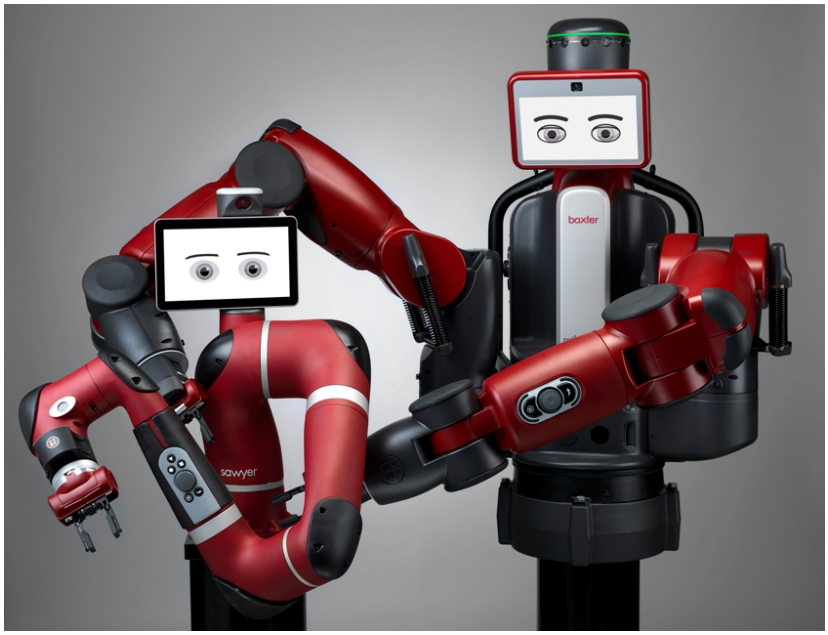


Figure A-3: **Sawyer And Baxter** (photo Jeff Green/Rethink Robotics).

developed these cameras and AI systems to recognise edges and shapes. By 1981 this research left the lab and headed to the industry where another General Motors factory implemented a system of vision called Consight [Holland et al. 1979]. Consight's system comprised of three different robots that used three vision systems to sort six different forms of automotive parts moving on a conveyor belt.

Some industrial robots of today are becoming more general purpose. Baxter is a humanoid industrial robot which is 2 metres tall and weighs 136kg and has a screen with a cartoon-like face, Figure A-3, to avoid the *'uncanny valley syndrome'* [Mathur and Reichling 2016]. Baxter uses the robot operating system (ROS), which is a framework for writing robot control software. The ROS contains

> *"a collection of tools, libraries and conventions that aim to simplify the task of creating complex and robust robot behaviour across a wide variety of robotic platforms."* [Quigley et al. 2009]

Today's robots include ubiquitous physical and verbal *"assistants"*. The grasping robot arm was one of the first steps [Levine et al. 2016], with motion planning [Pfeiffer et al. 2017] and visual navigation [Chen et al. 2015a], [Gupta et al. 2017]. The visual interaction with the outside world has led to the development of stabilised quadcopters [Zhang et al. 2016] and driving strategies for autonomous vehicles [Shalev-Shwartz et al. 2016] at which Google's Waymo and Tesla are at the fore-

Figure A-4: **(from left): HANDLE, SPOT and ATLAS Boston Dynamics Robots** (composite photo Boston Dynamics).

front. Boston Dynamics [Dynamics 2019], a robotics research company, recently purchased from Google by Tesla, have been developing animal and humanoid robots for some years, partially driven by Defense Advanced Research Projects Agencys (DARPAs) Robotics Challenge [DARPA 2015]. Boston Dynamics address three challenges with their robots:

- balance and dynamic mobility;
- mobile manipulation;
- mobile perception [Raibert 2017].

These robots have become very advanced, demonstrating their sensing and inspection abilities in the animal-like SPOT variant; their parkour abilities with their humanoid ATLAS variant; and their wheel-driven mobile manipulation machine called HANDLE, see Figure A-4. The robots movement, sensing and manipulation within the real world require AI to be successful.

A.1.2 Machine Learning (ML)

The term *Machine Learning* appears to be first coined in Samuel's work [1959], where Samuel proposes schemes for a machine to learn to play checkers to greater proficiency than that of the programmer. AI and ML are connected but distinct. ML is considered a subset of AI, as Figure A-5 shows. ML is a subset and a narrow form of AI, *i.e.*, we tailor the AI to a specific task. ML algorithms are used to collect and analyse huge amounts of data to enable the machine to 'learn', *i.e.*, identify patterns

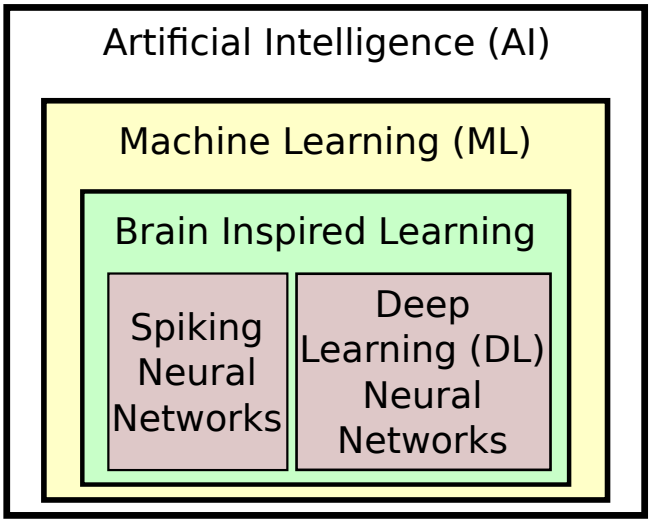


Figure A-5: **AI, ML, DL** [Sze et al. 2017].

and make intelligent decisions from the data, with little or no human intervention. A subset of ML is called deep learning (DL) and delivers common techniques that cover activities such as machine vision [LeCun et al. 1999], natural language processing [Collobert et al. 2011] and robotics [Sze et al. 2017], where hidden patterns can be learnt to arrive at a more efficient set of decision rules.

A.2 A Concise History of AI

How did we go about getting machines and automata to exhibit some form of *"intelligence"*? What does exhibiting intelligence mean? Today's significant developments in AI did not happen overnight. The innovation and development first started in the late 1930s, hitting two AI winters along the way before emerging as today's data-hungry Automata. A non-exhaustive list of the significant milestones achieved in AI is shown in Table A.1.

A.2.1 Highlights of Early AI Discovery, Innovation and Development

Alan Turing, in 1936, invented The Turing Machine [Church 1937], which is a mathematical model of computation to define an abstract machine. It would manipulate symbols on a strip of tape according to his proposed table of rules. He suggested that one could construct a Turing machine capable of simulating any given

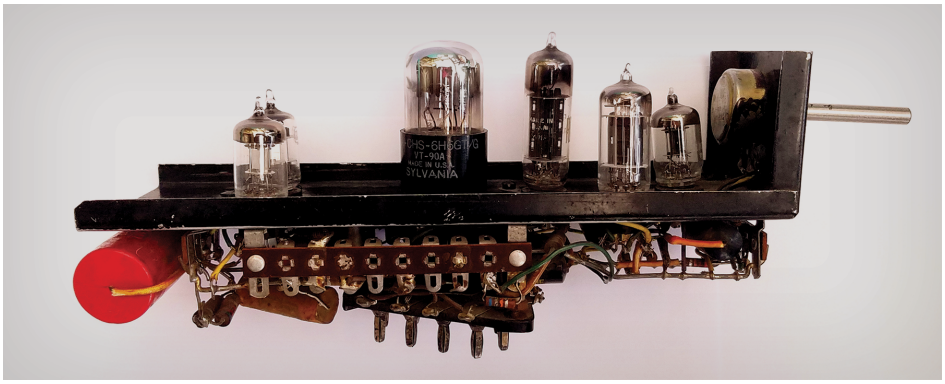


Figure A-6: **Stochastic Neural Analog Reinforcement Calculator (SNARC)** (photo Margaret Minsky).

computer algorithm, regardless of the machines simplicity or the complexity of the algorithm.

McCulloch and Pitts [McCulloch and Pitts 1943], proposed their Thresholded Logic Unit which shows how a neuron model with an activation function would fire and can be modelled mathematically using Boolean logic. Vannevar Bush's seminal work 'As We May Think' [Bush and Bush 1945], proposed a system which amplifies people's own knowledge and understanding. Alan Turing, five years later, wrote the article 'Computing Machinery and Intelligence' [Turing 1950], suggesting that machines could simulate human beings and undertake intelligent things such as play Chess.

Turing went on to develop his early work and proposed the 'Imitation Game' [Turing 1950]. The work suggests that if a computer could imitate human sentient behaviour, then this might imply that the computer *was* sentient! This idea has far-reaching implications. According to Turing, building a computer capable of passing the imitation game would require the computer to, for example, process natural language, learn from the conversation, remember what was said and the context, communicate the ideas back to humans and understand the implications of the communication. Turning also speculates that the imitation the computer might display might be interpreted as a form of common sense[1].

In 1956, John McCarthy, of Stanford and one of the godfather's of AI, coined the term artificial intelligence (AI). He coined the term when, along with Marvin

[1]Horace Greeley said, *"Common sense is very uncommon!"*. The opinion of the author is that if humans cannot display common sense, even though we might recognise it, how can we expect computers to exhibit common sense, especially as we are training them with our biased, nonsensical data?

Table A.1: Timeline Taxonomy of major milestones of AI, ML and DL.

1937	Turing's *"Turing Machine"*, mathematical model of computation [Church 1937]
1943	McCulloch and Pitt's Threshold Logic Unit (early neural network) [McCulloch and Pitts 1943]
1948	Shannon's *"A Mathematical Theory of Communication"* [Shannon 1948], pathing the way for modern digital communication
1950	Turing's *"Imitation Game"* asks can machines think? [Turing 1950]
1951	Minsky builds the first 40-neuron neural net machine, called SNARC, see Figure A-6
1956	McCarthy coins the term AI [McCarthy et al. 2006]
1958	Rosenblatt's Perceptron, a binary classifier proposed [Rosenblatt 1958]
1959-1961	Widrow's Adaline [Widrow and Hoff 1960], a single layer neural net proposed; Rosenblatt's multi-layer perceptron (MLP) proposed [Rosenblatt 1961]
1959	Lettvin probes frog's optic nerve to find patterns, not pixels [Lettvin et al. 1959]
1959	Hubel and Weisel probe cat's cortex to find neurons firing [Hubel and Wiesel 1959]
1965	Moore's law proposed; transistors to double in chips yearly [Moore 1965]
1969	Minksky *et al*., Perceptron Problem posed [Minsky and Papert 1969]
1974	Dennard proposes metal oxide semi-conductor field effect transistor (MOSFET) Scaling (referred to as Dennard Scaling) [Dennard et al. 1974]
1974	Werbos discovers backpropagation [Werbos 1974]
1982	Hopfield proposes the RNN [Hopfield 1982]
1985	Ackley *et al*., publish *"A Learning Algorithm for Boltzmann Machines"* [Ackley et al. 1985], proving multi-layered networks can learn, a counterpoint to Minsky's Perceptron Problem
1986	Rumelhart *et al*., publish *"Learning Internal Representations by Error Propagation"* [Rumelhart et al. 1985], introducing the BackProp algorithm used in DNNs
1986	Restricted Boltzmann machine (RBM) proposed [Hinton et al. 1984]; Navlab, the first autonomous car is produced [Thorpe et al. 1988]
1987	Madaline, the multi-layer Adaline [Widrow 1987]
1988	Sutton publishes *"Learning to predict by the methods of temporal differences"* [Sutton 1988] suggesting temporal difference learning is believed to be how human brains learn
1989	Mead publishes *"Analog VLSI and Neural Systems"* [Mead 1989], founding the field for neuromorphic engineering, building ICs inspired by biology
1989	Time-delay neural network, proposed [Waibel et al. 1989]
1989	The Connection Machine invented [Hillis 1989]
1989	First deep neural net to recognise digits, LeNet proposed [LeCun et al. 1989]
1992-1995	Support vector machine (SVM) and non-linear classifiers [Boser et al. 1992; Cortes and Vapnik 1995]
1997	LSTM [Hochreiter and Schmidhuber 1997]; Bidirectional recurrent neural network (BRNN) proposed [Schuster and Paliwal 1997]
1997	IBM's Deep Blue beats Kasparov at Chess [Kasparov and Greengard 2018]
1998	First ML dataset, MNIST [LeCun et al. 1998]
1998-1999	CNN with backpropagation, [Werbos 1974; Rumelhart et al. 1985; Lecun et al. 1998; LeCun et al. 1999]
2005	Thrun and his team win the DARPA Grand Challenge for an Autonomous Vehicle [Thrun and Montemerlo 2005]
2006-2007	Deep learning and deep belief network (DBN) proposed [Hinton 2002; Hinton et al. 2006; Hinton and Salakhutdinov 2006]
2007	First ML framework, Theano, introduced [Team et al. 2016]
2008	Delbruck develops spiking retina chip, dynamic vision sensor (DVS) [Delbruck 2008], which uses asynchronous spikes rather than synchronous frames used in cameras
2009	ImageNet dataset created [Deng et al. 2009]
2009	CIFAR-10/100 datasets created [Krizhevsky and Hinton 2009]
2011	First Hardware Neural Network accelerator released [Farabet et al. 2011]
2012	AlexNet [Krizhevsky et al. 2012], the 'one that started it all'; Dropout proposed [Srivastava 2013]
2013	ReLU and Dropout Works for speech recognition [Dahl et al. 2013]
2014	Generative adversarial network (GAN) [Goodfellow et al. 2014]; DeepFace [Taigman et al. 2014] introduced
2015	Tensorflow [Abadi et al. 2016] and Keras [Chollet 2015] frameworks introduced
2015	Intel Acquires FPGA company Altera [*Intel Completes Acquisition of Altera*]
2016	Pytorch framework released [Paszke et al. 2017]
2016	Alphabet's AlphaGo beats Sedol at Go and posits conjecture of a connection to Church-Turing Thesis [Wang et al. 2016b]
2016	Tesla Autopilot introduced in limited-access highways [Dikmen and Burns 2016]
2016	Intel Acquires AI startup Nervana Systems [*Intel Agrees to Acquire Nervana Systems*]
2016	Intel Acquires AI at the edge Irish startup Movidius Systems [*Intel to Acquire Movidius*]
2017	AlphaZero [Silver et al. 2017], Capsule Networks [Sabour et al. 2017] released
2017	Waymo's self driving car [Brown 2016]
2017	First ML ASIC accelerator, TPU [Jouppi et al. 2017]
2016-2019	DeepFakes [Thies et al. 2016; Suwajanakorn et al. 2017] seen in the wild
2018	Bidirectional encoder representations from transformers (BERT) [Devlin et al. 2018], 10% improvement in search
2017-2018	Unsupervised machine translation [Lample et al. 2017] proposed
2019	Alphabet's AlphaStar beats humans at Starcraft II [Arulkumaran et al. 2019]
2019	OpenAI's robot hand solves Rubik's Cube [Akkaya et al. 2019]
2019	OpenAI's AI plays Hide-and-seek games, breaks physics [Baker et al. 2019]
2019	Boston Dynamics release SPOT, ATLAS and HANDLE robots [Dynamics 2019]
2019	Intel Acquires AI chipmaker Habana Labs [*Intel to Acquire Artificial Intelligence Chipmaker Habana Labs*]

Artificial intelligence (AI) — Machine learning (ML) — Deep learning (DL) — 1st AI Winter — 2nd AI Winter

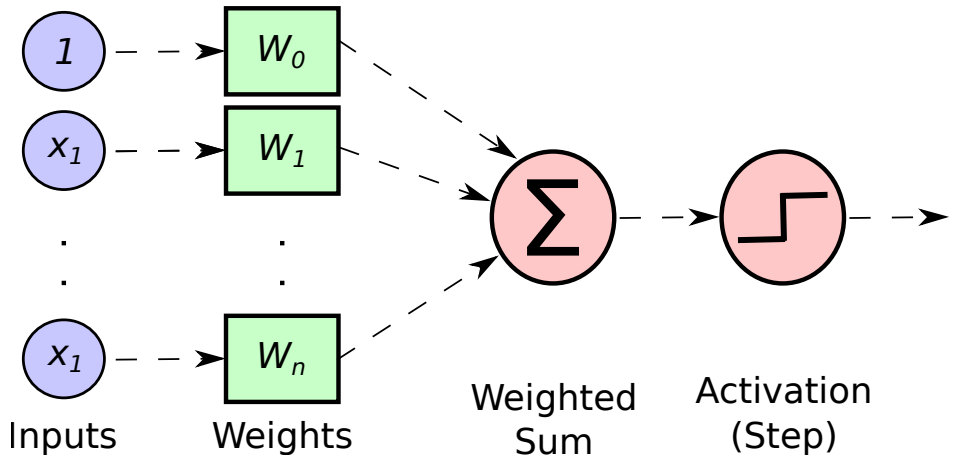


Figure A-7: **Perceptron** [Sharma 2017].

Minsky of Harvard University, held the first academic conference called the 'Dartmouth Summer Research Project on Artificial Intelligence Workshop' [McCarthy et al. 2006].

A.2.2 The Perceptron and Multi-layer Networks

Building on McCulloch and Pitts 1943 work, Rosenblatt, in 1958, proposed the Perceptron [Rosenblatt 1958], which is a supervised learning algorithm for binary classifiers, something upon which some of today's DNNs rely. A binary classifier's function determines if an input is categorisable into classes to determine predictions based on a linear predictor function, which combines a set of weights with the feature vector. The activation or transfer function receives the output and adjusts the weights during learning and is usually a step function in the Perceptron, see Figure A-7.

By 1959, Lettvin *et al.*, placed probes into the optic nerve of a frog from which they worked out that the signal that was being sent from the eye to the brain was not just a pixel, it was a pattern that could detect a black dot moving across a white background [Lettvin et al. 1959]. The nerve cells of the eye were encoding information that was passed to the brain. At the same time, Hubel and Weisel in 1959 [Hubel and Wiesel 1959] were trying to understand how the visual cortex works by studying how neurons fired in a cat's cortex. Their discoveries won them the Nobel Prize in 1981. The neuron work was essential for illuminating our understanding of the visual system and supplying the framework for modern-day neural nets. From

this work, they found that similar to the cat's striate cortex the neuron models can detect edges of objects.

In 1960, Widrow and his graduate student Hoff took McCulloch's Perceptron idea further. Widrow *et al.*, produced a network of nano electric circuit elements called *"memistors"*, an ad hoc 3-terminal device-specific to one application, and not to be confused with memristors (see Kim *et al.*, for clarification [2012]). During the learning phase, the memistor would adjust the weights according to the weighted sum of its inputs. In comparison, the Perceptron passed the weighted sum onto the transfer function. The researchers called these single-layer networks an Adaptive Linear Neuron or Adaptive Linear Element (ADALINE) [Widrow and Hoff 1960]. Widrow, in 1987, produced a multi-layer version of the network and termed it MADALINE [Widrow 1987] and appeared at the first IEEE International Conference on Neural Networks.

A.2.3 Amdahl's Law And Its Effect on AI

The work of Adaline and Madaline was proposing increased parallel computation. Amdahl proposes a formula to show the theoretical parallel speedup of a program, and its execution latency is a function of the number of processors executing the program [Amdahl 1967]. He suggests that the serial portion of the program limits the speedup. He shows in Equation 1 that the speedup of a parallel system is:

$$Speedup = \frac{1}{1 - p + \frac{p}{n}} \tag{1}$$

where:

p = proportion of the algorithm that can be parallelised;

n = number of CPU cores or threads.

So for a program that runs for *e.g.*, 20 hours on a single processor core or thread, if a non-parallelisable portion of that program takes one hour to execute, then the whole program cannot take less than one hour, regardless of how much more of the remainder of the program is parallelised, see Equation 2. That is:

$$Speedup = \frac{1}{1 - 0.95 + \frac{0.95}{2048}} = 19.82 \tag{2}$$

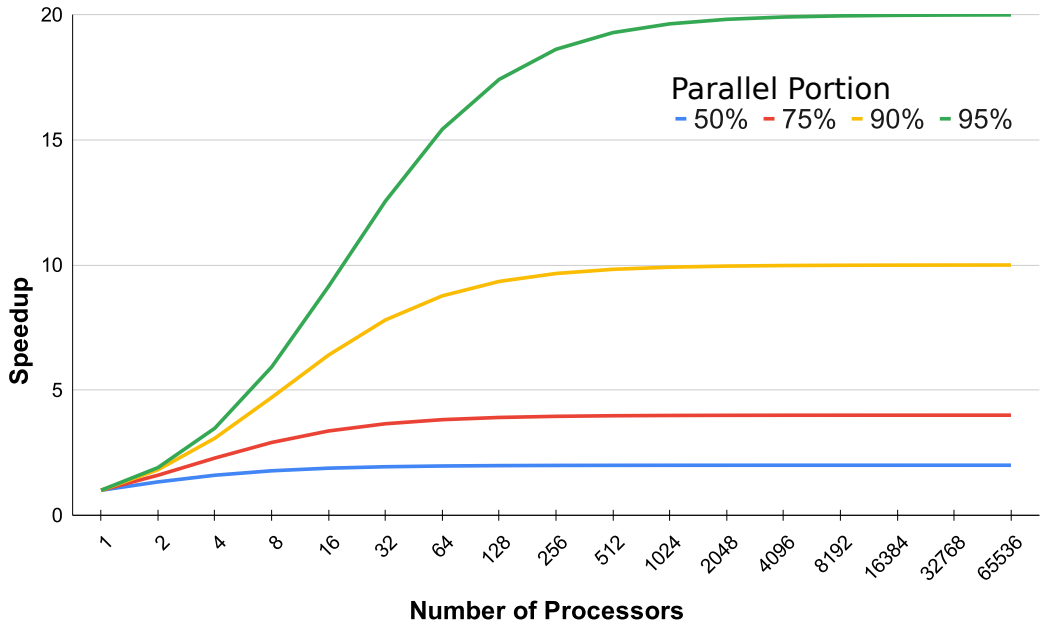


Figure A-8: **Amdahl's Law** [Amdahl 1967].

where:

$p = \frac{19}{20} = 0.95$ and number of cores = 2048

Thus, Amdahl proposed, the theoretical speedup is, in this example, limited to $20\times$. Therefore parallel processor systems would only be used for highly parallelisable programs, see Figure A-8. This limit might suggest a bottleneck for the progress of AI systems (more on this in subsection A.2.6).

A.2.4 The XOR Problem and the Dawn of the First AI Winter

XOR logic is a gate with two binary inputs and one output, the output of which should return and true value if the two inputs do not match. One would think that this would be a simple function for a machine to learn. However, in 1969 Minsky and Papert [Minsky and Papert 1969] showed that Perceptrons are incapable of learning the simple XOR function due to the Perceptron needing the output functionality to be linearly separable into correct classification categories, something an OR gate has but an XOR gate does not. The XOR problem meant many researchers lost interest in the field of AI and so ushered in the first AI winter, where research in the area dwindled. One researcher, Werbos, was not put off and in his 1974 dissertation, [Werbos 1974] described the training of neural networks through a process of backpropagating the error into the network. The collaborative backpropagation

work of [Rumelhart et al. 1985] in 1985 that built on Werbos work. Ackley *et al.*, , in the same year [Ackley et al. 1985], published work that proposed a solution to the XOR problem that help neural networks learn non-linear concepts. Both the XOR solution and the backprop were two key works that helped bring the field out of the first AI winter.

A.2.5 Moore's Law and Dennard's Scaling

As research in AI increased again in the late 1980s, a requirement for faster microprocessing was required. Gordon Moore, who founded Fairchild Semiconductor, in his 1965 research proposed that transistors in a microchip would double about every year [Moore 1965] which he later revised to every 18 months. Robert H. Dennard co-authored work in 1974 [Dennard et al. 1974] that suggests for every new technology generation of transistors, as the geometry of the transistor decreases, the energy consumption will stay constant for twice the number of transistors if the transistor density doubles. In turn, the circuit shall become 40% faster and 30% smaller. Dennard *et al.*, called this MOSFET scaling, but it would later become known as Dennard Scaling. Moore, in 1975, revised his law to state that transistor count would double every two years. When combined, Moore's law and Dennard scaling show that performance per watt grows even faster, doubling about every 18 months.

A.2.6 Connection Machines

In 1983, during his PhD studies, Danny Hillis, under his adviser, Sussman, with input from Minsky and Shannon, founded the company Thinking Machines Corporation with Sheryl Handler. To Hillis, it is evident that the brain worked much faster than computers, and that if computers were even to get close to matching that of the brains capability, the computer would have to have a similarly massively parallel architecture. Hillis challenged Amdahl's notion was of a theoretical parallel processor limit. He demonstrated that as machines got more substantial, the machines could solve more extensive problems with more data on which to work [Hillis 1989]. Hillis demonstrates that the speedup does not peak. The peak can be increased even as the problem, and thus, the parallelism increases. Hillis shows how a separate processor and memory element could process each pixel of an image producing the first two-dimensional processor grid at a micro-architectural level

and built this parallel processor to demonstrate the fact. The CM-1 super-computer that Hillis built contained 65,536 processors that processed one-bit at a time but in a SIMD manner. For comparison, at the time, Cray had the fastest super-computers in the world. In comparison to the CM-1, the Cray computers were deeply pipe-lined, fast switching, clocked as high as possible, and liquid-cooled. The wire tracks of processors chips were made as short as possible to reduce the latency of the signals in the processor and between four and eight parallel processors were implemented. A year later, Hillis' company built the CM-2 which upgraded the CM-1's fixed-point computation to FP with its floating-point co-processor and added more RAM and a larger hard drive. Geoffrey Hinton, a Turing Award-winning AI researcher, was one of the first people to use the CM-2 machines for his connectionist algorithms [Hinton 1989].

A.2.7 The Second AI Winter

In the late 1980s, interest and funding in AI waned again, forcing the area to enter a second AI winter which may have been partly due John McCarthy's 1984 criticism of expert systems and their lack of common sense and knowledge about their limitations. The collapse of the Lisp market may have also contributed. Jack Schwarz, Director of DARPA/ISTO (Defense Advanced Research Projects Agency/Information Science and Technology Office) from 1987 to 1989 argued that AI had yet to demonstrate:

> *"any unifying principles of self-organisation,"*

meaning that its

> *"applications must still be seen as adaptations of diverse ideas rather than as systematic accomplishments of a still mythical AI technology."* [Roland et al. 2002]

A.2.8 The Advent of Modern-Day Vision Machine Learning

Hubel and Wiesel's work [Hubel and Wiesel 1959] on the visual cortex in mammals was Fukusima *et al.'s*, inspiration for the Neocognition work [Fukushima 1980] Figure A-9, which could work for some aspects of computer vision. Neocognitron

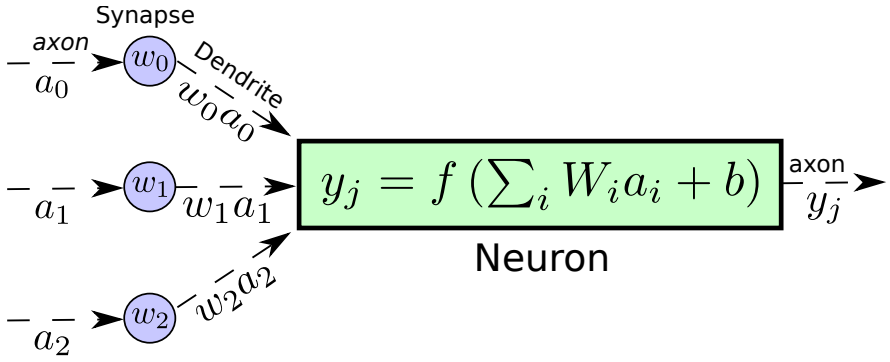


Figure A-9: **A Model of a Brain Neuron** a_i (activation), w_i (weight), $f(.)$ (non-linear function), b (bias). [Li et al. 2017b].

was made from a set of layers. An input layer is presented with a stimulus, for example, the pixels in a digital image. Subsequent layers would analyse the image, looking for variations, in contrast, edges, ultimately culminating in an output layer that would categorise the group to which the image belongs. Connections between each layer allow for all the relevant processing to take place.

In the late eighties and nineties, Waibel *et al.*, [1989] and separately LeCun *et al.*, [LeCun et al. 1989; Lecun et al. 1998; LeCun et al. 1999] took this work further to propose CNNs with backpropagation. Backpropagation is used to train supervised (data is labelled) feedforward networks. Improved computation coupled with the large amount of large data available from the postal service in the USA allowed LeCun *et al.*, to apply their research to Zip code recognition on envelopes and packages.The mid-to-late 1990s with backpropagation in CNNs and LSTM, a type of reinforcement learning (RL), marked the beginning of the end of the second AI winter.

A.2.9 ImageNet

Deng and his colleagues at Princeton, under Processor Li Fei-Fei generated ImageNet [Deng et al. 2009]. ImageNet is an extensive visual database designed for image recognition. More than 14 million images are hand-annotated with information about the contents of the images. More than one million of the images provide

Table A.2: ILSVRC Winning CNNs.

Year	Model	Top-1 Error	Top-5 Error	Total Layers	MAC	Number of Parameters
2012	AlexNet [Krizhevsky et al. 2012]	$\approx 36.7\%$	$\approx 15.4\%$	8	724M	$\approx 62M$
2013	ZFNet [Zeiler and Fergus 2013]	$\approx 36\%$	$\approx 14.7\%$	5	n/a	n/a
2014	VGG-16 [Simonyan and Zisserman 2014b]	$\approx 25.6\%$	$\approx 8.1\%$	16	154.2G	$\approx 138M$
2014	VGG-19 [Simonyan and Zisserman 2014b]	$\approx 25.5\%$	$\approx 8\%$	19	n/a	$\approx 144M$
2015	Inception V1 [Szegedy et al. 2015]	$\approx 30.2\%$	$\approx 10.1\%$	27	1.43G	$\approx 7M$
2016	ResNet-152[He et al. 2015]	$\approx 19.4\%$	$\approx 3.6\%$[2]	152	11.3G	n/a
2017	SeNet-154 [Hu et al. 2018]	$\approx 18.7\%$	$\approx 4.5\%$	154	3.87G	$\approx 440M$

bounding boxes around the annotated contents. Without this dataset, training of CNNs would prove difficult.

A.2.10 ILSVRC and Kaggle Competitions

The ILSVRC competition [Russakovsky et al. 2015], which started in 2010 covered the following tasks:

- Classification: classify an object in an image;
- Localisation: localise the classified objects in the image with bounding boxes;
- Detection: detect an object and assign a top-5 prediction or predict the background class when there are no objects detected.

The winners of the ILSVRC between 2012 and 2017, before Kaggle took ownership of the competition, are listed in Table A.2. The trend in improving *top-5* and *top-1* error of the models often means an increase in the number of parameters and layers. The increase in layers and parameters suggests larger compute and bandwidth requirements, although not all works quote these operational or bandwidth performance.

Researchers focus on optimisations that training of CNNs is performed on larger models with more parameters, which in turn exhibit lower error or better classification accuracy of the CNN. More extensive models, *e.g.*, ResNet with its thousand-plus layers, have several inherent problems such as compute and memory bandwidth. The next section will highlight research areas and work that attempt an algorithmic optimisation, regardless of the hardware or software platform implementation.

[2]Human Beings' Top-5 Error Rate is $\approx 5.1\%$

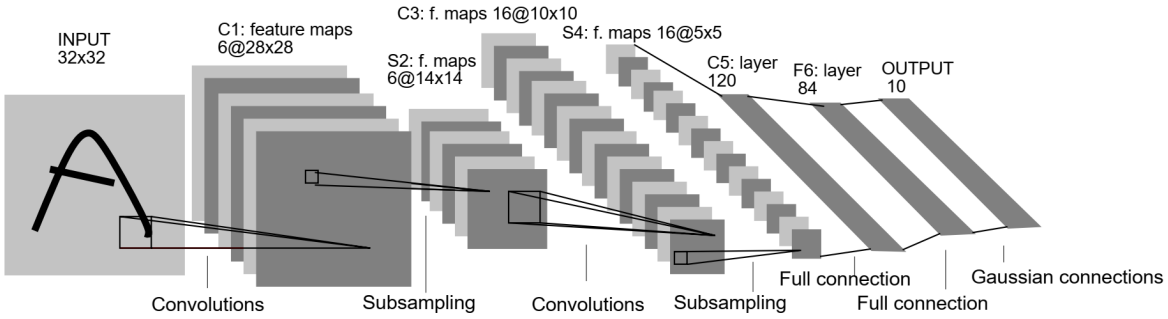


Figure A-10: **LeNet's Architecture.** [LeCun et al. 1989].

A.3 Most Influential CNN Models

Researchers often build on the work from LeCun *et al.*, and their LeNet [1998; 1989] CNN, that has since been widely used by the postal services to recognise handwritten postal addresses. LeNet network consists of:

- two 5×5 convolution filters which are applied at a stride of one to the input;
- two sub-sampling or pooling layers (*Pool* below) of a 2×2 configuration applied at stride two;
- two fully connected layers (*FC* below) to give the output, all connected in the sequence (see Figure A-10).

For each algorithmic optimisation, the following subsections highlight how the classification accuracy, area, energy, performance and storage are affected and how the layer configuration is compared to the above LeNet version. Typically the latter work cover areas of data types and sparsity of the weights or network.

CNNs perform with optimal classification accuracy of inference when computing using 32-bit or 64-bit FP arithmetic [IEEE 2019]. To increase performance and throughput of FP arithmetic, a hardware FPU is typically implemented on the same IC die as the CPU.

Krizhevsky *et al.*, in their AlexNet [2012] network, implement five convolutional layers and three fully-connected layers in a different configuration to LeNet. They adopt the configuration as shown in Figure A-11.

They connect the last fully-connected layer output to a 1000-way softmax exponential function layer which corresponds to 1000 class labels in the ImageNet dataset. Between each convolution layer the researchers employ ReLU as their non-linear activation function instead of the hyperbolic tangent function TanH that

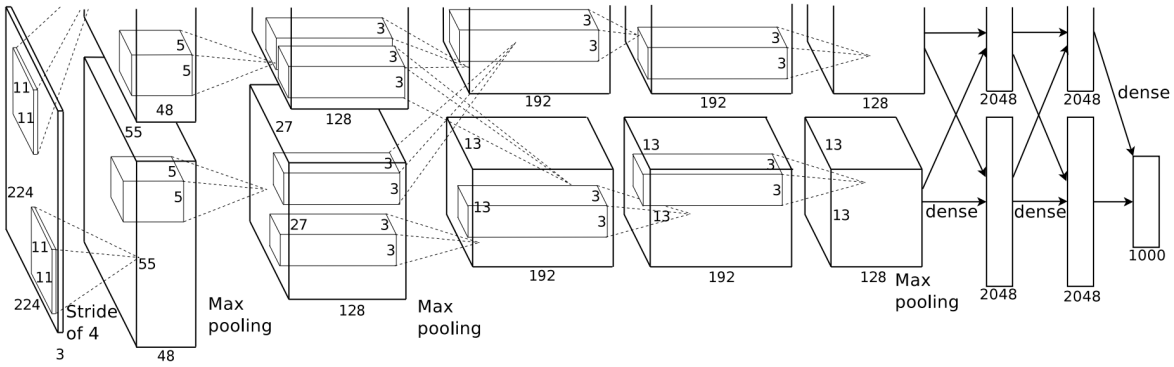


Figure A-11: **AlexNet Architecture, split across two GPUs** [Krizhevsky et al. 2012].

LeNet [LeCun et al. 1989] employs (see Figure A-10 for LeNet's architecture). ReLU helps to fix the vanishing gradients problem that Sigmoid and TanH functions exhibit, see subsection B.2.4 for more in-depth discussion. AlexNet exploits overlapping maximum pooling to prevent over-fitting. AlexNet also uses "dropout" which zeroes the output of each hidden neuron with a probability of 0.5. The "dropped out" neurons do not contribute to the forward pass or backpropagation. These algorithmic changes improve the Top-1 classification accuracy of 14 million images belonging to 1000 classes from the previous ILSVRC years' Sift and Fisher Vectors of 45.7% to the record-breaking and 2012 ILSVRC winning 37.5% and a top-5 error rate of 15.4%. AlexNet requires a large amount of training data, such as that supplied by ImageNet [Deng et al. 2009]. When pretrained, AlexNet has up to 240MB of trained kernel weight data.

ZFNet [Zeiler and Fergus 2013] demonstrated an 11.7% top-5 error rate. This research also demonstrated a visualisation technique, see Figure A-12. Figure 2 of the work aids the understanding of what each filter in each layer of AlexNet was trying to detect in an image, *i.e.*, edges or lines of items in an image, or certain high-level features like letters of a word, faces or eyes, essentially a much more detailed version of the visualisation of AlexNet.

In 2014, VGG-16 [Simonyan and Zisserman 2014b], shows that VGG-16 has increased the layer count to between sixteen and nineteen layers in comparison to AlexNet's eight layers. However, they only employ convolution with 3×3 filters at a stride of one, and a padding of one (also known as *"same convolution"*). Their max pooling is 2×2 with a stride of two, see Figure A-13.

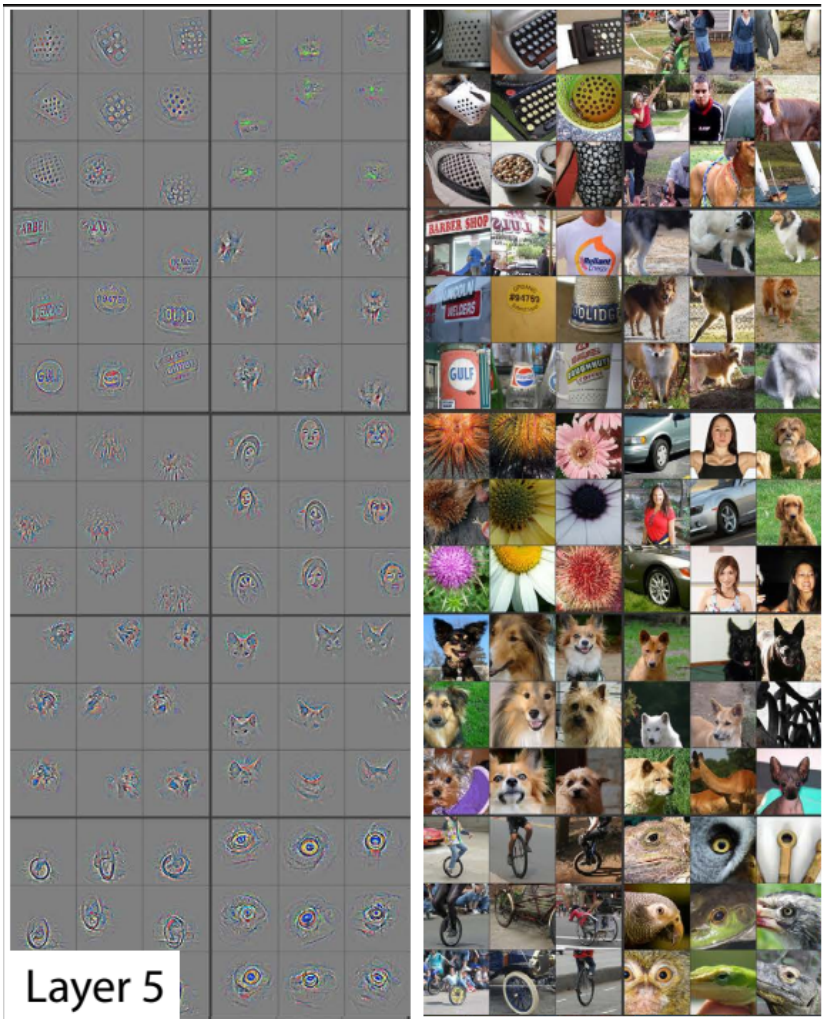


Figure A-12: **Snippet of Figure 2 of the work to demonstrate the faces and eyes visualisations of layer 5.** [Zeiler and Fergus 2013].

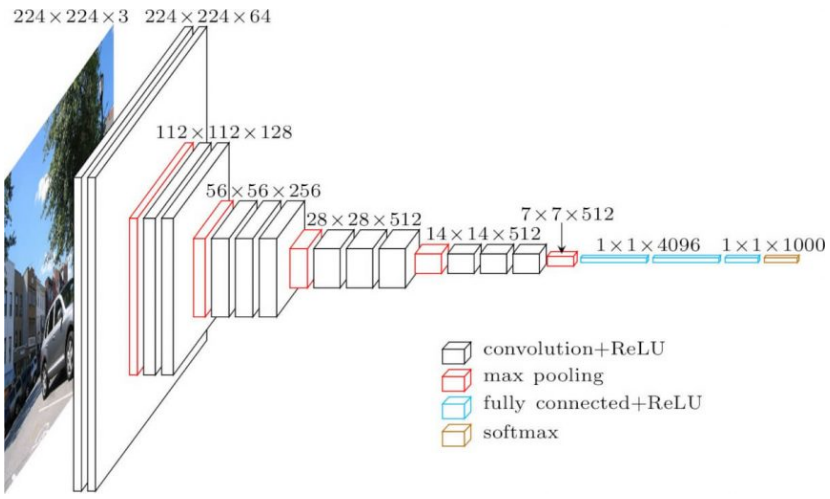


Figure A-13: **VGG-16 Architecture (image by Frossard).** [Frossard 2016].

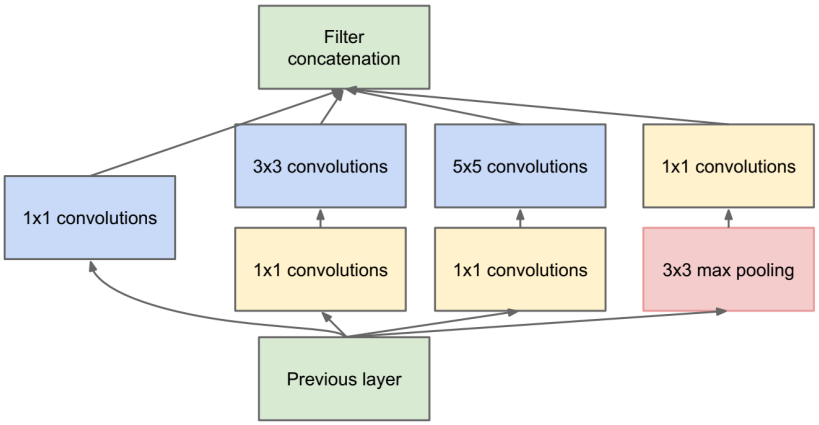


Figure A-14: **GoogLeNet's Inception Module.** [Szegedy et al. 2015]

Simonyan *et al.*, find that a stack of three 3×3 convolution layers has the same effective receptive field as one 7×7 convolution layer. VGG-16 is, therefore, a deeper network with more non-linearity units between the convolution layers. While it may be deeper, VGG-16 contains fewer parameters with:

$$k * (k^2 C^2) \tag{3}$$

where $k = 3$ compared to:

$$k^2 C^2 \tag{4}$$

where $k = 7$ for ZFNet for C channels per layer.

Google's GoogLeNet [Szegedy et al. 2015] further deepens the network. The computational complexity is more straightforward than the previous years' ILSVRC contestants. There are no multiple, expensive FC layers at the output and the whole network contains approximately $12\times$ fewer parameters than that of AlexNet, demonstrating a top-5 classification error rate of 6.7%.

Google achieved this by proposing layers of Inception Modules, that perform dimensionality reduction, and they claim also reduced computational complexity. The inception module is designed with network topology in mind, see Figure A-14, and applies parallel filter operations and pooling from the previous layer. The output of these operations is concatenated. To reduce the computational complexity, Google applies *"bottleneck"* convolution units with filters. They claim the bottleneck, and convolution units with filters preserve spatial dimensions and reduce the depth while projecting the depth to lower dimensions in the form of a combination of

feature maps. Google add small stem networks before the inception modules and an auxiliary classification layer to inject additional gradients at the lower layers, see Figure A-15.

ResNet [He et al. 2015] proposed by Microsoft's He *et al.*, further increases the layer count. ResNets can range from 18- to 152-layers. They demonstrate a 152-layer version applied to ImageNet reaches a top-5 classification error of 3.57%, which is better than human performance [Russakovsky et al. 2015]. Their research asked what would happen if the number of layers in a network is continually increased. If the increase in layers is performed naively, they show that both the training and test error increase, but over-fitting does not cause the error. They go on to hypothesise that this is due to an optimisation problem as deeper models are harder to optimise.

They found that a solution is to copy the learned layers from the shallower model and set additional layers to identity mapping. In other words, they found that instead of trying to fit a desired underlying mapping directly, a better approach is to use network layers to fit a residual mapping.

The ResNet learning block in Figure A-16 shows that the researchers use layers to fit residual $F(x) = H(x) - x$ instead of $H(x)$ directly as in a plain-layers version. These residual blocks have two 3×3 convolution layers. The Learning Block is ResNet's attempt to tackle backpropagation's vanishing gradient problem of very deep models. Vanishing gradients occur when the element of the gradient becomes exponentially small as the updates flow backwards to the initial layers of the model. The update to the initial layers becomes increasingly small and thus increasing training time and reduces convergence. The skip connections of the Learning Block allow the gradient to flow back to the initial layers unhindered.

Figure A-17 shows that to create the full network, the researchers stack residual blocks. Periodically through the layers, there are double the number of filters applied and downsampled spatially using a stride of two, *i.e.*, divided by two in each dimension. The researchers add a convolution layer at the beginning of the network. They apply a global average pooling layer after the last convolution layer. Notice that there are no FC layers at the end, only a FC-1000 to output the classes. Similar to GoogleNet, ResNet adds a bottleneck layer, Figure A-18 to improve efficiency, for ResNet-50 layers and above.

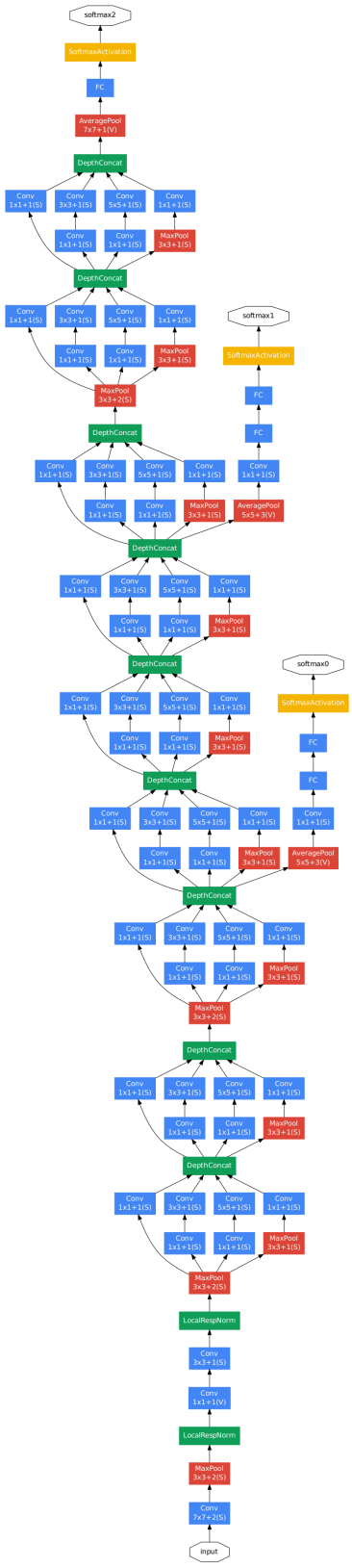


Figure A-15: **Complete GoogLeNet.** [Szegedy et al. 2015].

Howard *et al.*, proposed MobileNets [Howard et al. 2017] which is a different take on the convolution schemes of other networks such as GoogleNet. They replace

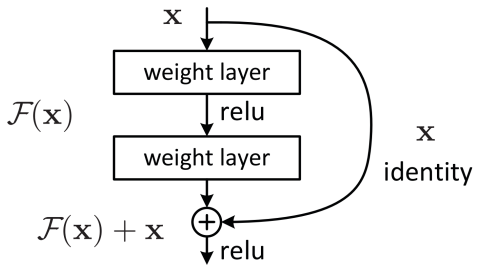


Figure A-16: **ResNet Learning Block.** [He et al. 2015].

regular convolution filters with depthwise convolution, followed by 1×1 pointwise convolution filters. They call this combination a depthwise separable convolution. They configure a 30-layer model, the first layer containing a regular convolution followed by thirteen layers containing depthwise, followed by pointwise convolutions with differing stride values. The model finishes with an average pool, fully connected layer and a softmax classifier.

When compared to a full convolution, in their experiments MobileNets depthwise separable convolution only loses about 1% classification accuracy but has approximately 10% of the multiply-adds operations which would otherwise account for a large amount of compute and latency. Their Stanford Dogs training [Khosla et al. 2011] classification accuracy is 83.3%, marginally behind that of GoogleNet. However, their computational complexity and storage are far less than GoogleNet (see Table 10 of [Howard et al. 2017] for more details).

A.4 Challenges of Deep Learning

The above history has pointed out some of the challenges researchers have had to surmount to get DL to where it is today. There seem to be three main reasons why researchers and engineers find DL so hard to implement:

1. Large datasets are required;

    - DL requires large amounts of data that has to been cleaned and often preprocessed in order to generalise the large feature hierarchies;

2. A Large amount of computational energy is required;

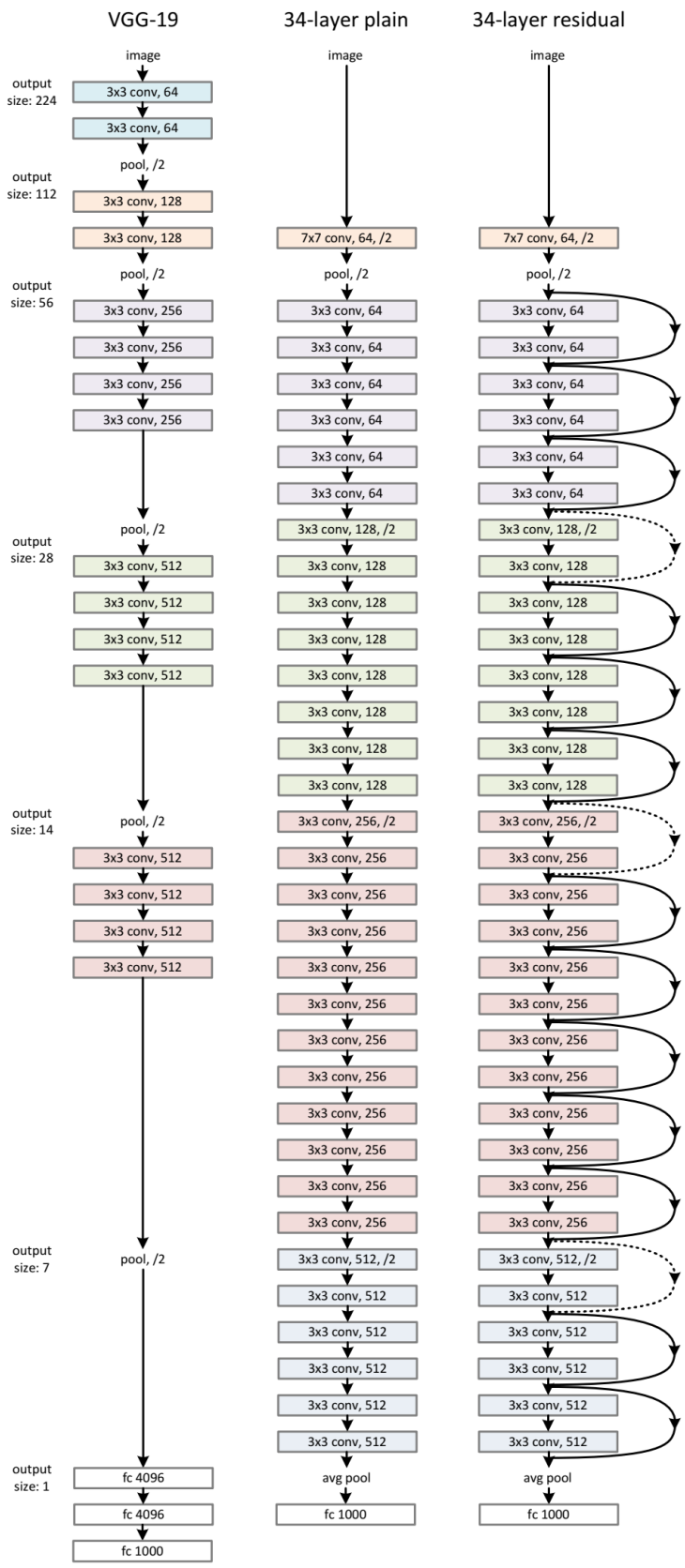


Figure A-17: **ResNet Example.** [He et al. 2015].

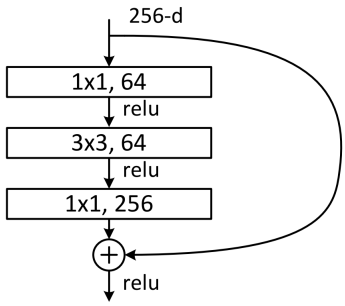


Figure A-18: **ResNet Bottleneck Layer.** [He et al. 2015].

- For the above large dataset, a large number of computations will be required to iterate over the training data in order for learning to converge, while not overfitting the test data;

3. It is hard to train and optimise DL algorithms;

   - Unless the features of the model that are being learnt are known ahead of time, it is difficult to know the reasons that DL algorithms are failing to converge.

A.5 Conclusion

As can be seen from the ILSVRC winners, Table A.2, the trend of improving performance of the CNNs is in part due to the massively parallelised CNN implementations, larger CNN models and datasets and not due to scaling with Moore or Dennard's laws. Furthermore, to sum up, some of the achievements of neural networks, LeCun, Bengio and Hinton[3]discuss how deep learning allows computational models to learn representations of data at multiple levels of abstraction [LeCun et al. 2015] in existing hardware. They show how these methods have dramatically improved speech recognition, vision object recognition and detection in all sorts of domains from drug discovery to genomics and automated vehicle driving.

We have highlighted the significant milestones in the history of AI, DL and DNNs. Major obstacles have been overcome over the last 90 years to get AI to the point today of advanced robotics, speech and image recognition and gameplay. We

[3]LeCun, Bengio and Hinton: 2019 Turing Award recipients, the Nobel Prize of computer science.

have highlighted various areas for comparison of the research works in order to identify common themes of investigation or gaps in the research.

B Introduction to Convolutional Neural Networks (CNNs)

B.1 Introducing Neural Networks

CONVOLUTIONAL NEURAL NETWORKS are key to modern-day AI, as can be seen from the extensive and wide-ranging applications highlighted from the articles in chapter 2 beginning on page 9. However, how exactly does a CNN work and what can be done to make them perform faster and more efficiently? This chapter will detail simple CNNs, delving into the arithmetic of how the CNN can recognise a digit in an image, and then build on this underlying understanding to show how a naive CNN does a similar task of recognition. From there, the chapter suggests where optimisations of these CNNs might occur and introduce how this thesis research tries to address how it tackles these optimisation needs.

B.2 Artificial Neural Networks and Machine Learning

In Appendix A beginning on page 103 we discuss how modern AI has had several highs and lows since the 1940s, but it was McCulloch and Pitt's Threshold Logic Unit [McCulloch and Pitts 1943] that convinced humans to believe the dream that computers could one day achieve human-level abilities. The 1970s and 1980s saw two AI winters where research slowed almost to a halt. The 1990s brought a Chess champion beating machine [Kasparov and Greengard 2018], and by the 2000s we saw the dawn of the modern voice and image recognition in neural networks, most notably demonstrated in the 2011 Jeopardy game show winning Watson AI machine from IBM. Since then, the pace of AI in academia and industry has accelerated, to the

point where AI and its subset ML are touching many aspects of our lives. Manufacturing and production [Li et al. 2017a], image tagging of our photos [Bambha et al. 2011], diagnosing medical conditions [Kapil et al. 2018] and prescribing appropriate drugs or interventions [Ferrucci et al. 2013], NLP and understanding [Cambria and White 2014] to speech synthesis [Ning et al. 2019] for communication back to us by our Google Home and Amazon Alexa and self-driving vehicles all have become accepted norms in our lives. How does AI work, and why has it become so ubiquitous?

B.2.1 Categories and Applications of ML

First, let us drill down into AI to its subset of ML to investigate the categories and example applications of ML. There are four major categories of ML models: supervised; semi-supervised; unsupervised; reinforcement learning, and very briefly they work as follows:

- **Supervised:** The ML model learns to predict the output of a model from the input after being trained with a labelled data set. Very briefly this is done as follow:
  - Supply input data and a labelled data set and train your ML model (see subsection B.2.5 for more details on training a supervised CNN);
  - Supply some unseen data to make:
    - a classification (*e.g.*, there is a cat in the image) [Krizhevsky et al. 2012] or object detection (*e.g.*, a dog is in the bottom left of the frame) [Redmon et al. 2016];
    - a regression (*e.g.*, house price prediction [Varma et al. 2018]).
- **Unsupervised:** The ML model learns the structure of the unlabelled input data as follows:
  - Supply your unlabelled input data to the ML model. The model shall:
    - group or cluster the data (*e.g.*, group images that contain dogs together and those that don't in separate groups);
    - associate certain data with features or activities (*e.g.*, user X on the streaming video service Netflix watched 'Y' show(s), therefore they will probably like 'Z' show(s) [Koren 2009]).

- **Semi-supervised:** The ML model leans from both labelled and unlabelled data:
  - Supply a small set of labelled data to train from (*e.g.*, a small set of hand labelled computed tomography (CT) scans of Covid-19 patients [Fan et al. 2020] or cancer biopsies [Kapil et al. 2018]);
  - The model expands on the small dataset to extrapolate its training set from which to learn;
  - GANs [Goodfellow et al. 2014] are a good example of semi-supervised ML. Among other tasks, GANs can generate faces that have never existed [Bao et al. 2017].
- **Reinforcement learning:** The ML model learns by being given rewards for outcomes:
  - Place ML agent in an environment and give it an action (*e.g.*, where are my glasses) and an expected outcome (*e.g.*, I can see where they are). A reward, positive or negative is given on some distance from the actual outcome;
  - The training is iterated until the ML arrives at the correct outcome (if ever) (*e.g.*, training a car to drive on the road [Thrun and Montemerlo 2005]. Do not hit pedestrian humans!)

So from the major categories above, choosing an ML model for a dataset is application-specific. One chooses what type of ML model is required based on the required outcome, e.g. prediction or clustering! However, ML is not always the perfect choice and can be overkill for applications that require simple data analytics. Now we have an overview of the types and applications of ML, we shall now look in more depth at the inner workings of ML models.

B.2.2 The Brain and Vision

To understand better how neural networks (NNs) work, let us start with the brain and see how it has inspired today's ML. The basic unit of a human brain is the neuron (or nodes), and the synapses connect the neurons, see Figure A-9. The brain consists of around eighty-six billion neurons and approximately one and a half trillion synapses [Azevedo et al. 2009]. These neurons and synapses can do amaz-

ing things to process our five primary senses, automating movement, breathing and making sounds, all within seconds of birth and with no prior training. The adaptability and plasticity of the brain allow humans to do very generalised learning from a very young age. Even one of the simplest forms of life with a whole connectome, the hermaphrodite nematode worm, has 302 neurons and 7446 synapses [Cook et al. 2019] yet can search for food and burrow unaided. So it is no wonder why researchers take inspiration from the brain to model its capabilities in a computer.

Let us focus on a subset of ML called deep learning (DL) and apply it to the field of vision processing. To apply DL to vision processing, we much first try to understand how human vision works at a high-level. When we see a digit, *e.g.*, 3, our brains can recognise it as a 3 and not an 8 or the letter E. The visual receptors of the eye would respond differently to a handwritten 3 from a printed 3, yet enough neurons and synapses of the visual cortex in our brain still recognise both as 3. To write a computer algorithm heuristically, that can recognise any number 3, in whichever manner it is written or printed, becomes a significant task for a computer scientist, and certainly does not scale well to larger recognition tasks.

B.2.3 Deep Learning

Researchers build a DNN and automatically train it with lots of different examples of *e.g.*, printed and handwritten 3 digits. The *"deep"* aspect of the name DNN refers to the number of hidden layers the DNN contains, see subsection B.2.4. The DNN is allowed to *'learn'* the configuration parameters of the DNN for a 3 digit. The trained DNN is then run in inference mode to infer the recognition of a new printed or written 3 in any context, to within the desired degree of prediction accuracy.

So how does this happen at the lowest level in the NN? Modelled on neurons and synapses of the human brain, see Figure A-7, NNs are a set of algorithms trained to recognise patterns in data. The NN is trained using a form of perception to detect, cluster and classify input data. The NN captures the patterns from real-world data, such as images from a camera, sounds from a microphone, text from a keyboard or time-series data from speech, and convert the data to a numerical format for the NN to process, usually employing a DSP. These example images, sounds or text

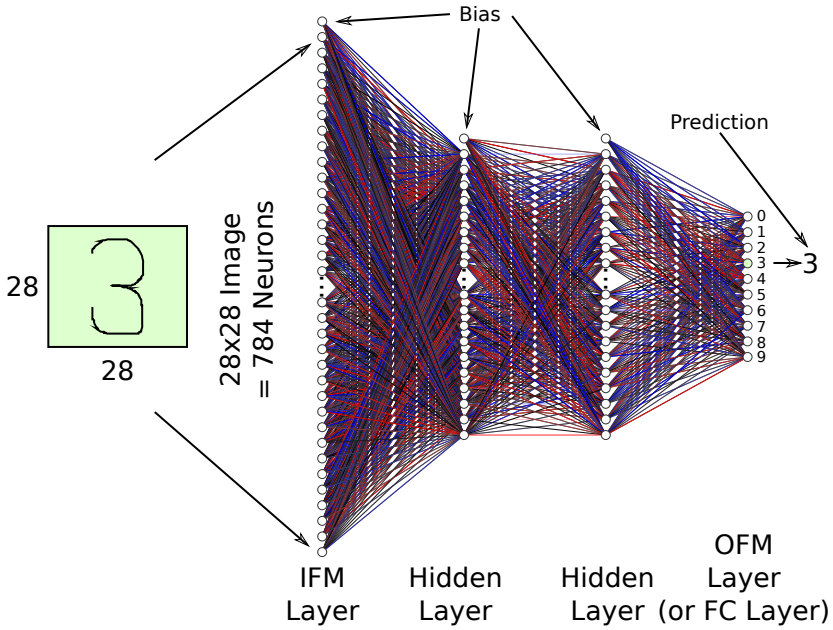


Figure B-1: **Multi Layer Perceptron.**

have features within them, *e.g.*, faces in the images, that the NN detect, classify and recognise.

B.2.4 Artificial Neural Network Model

To demonstrate how a artificial neural network (ANN) works in more detail, let us consider how one of the simplest ANNs called the multi-layer perceptron, Figure B-1, might recognise the digit 3.

A handwritten digit 3 is drawn on a grid of $28 \times 28 = 784$ pixels (left of Figure B-1), and each pixel represents a neuron. Assume each of the neurons holds a grey-scale FP value of that pixel called an *activation*, *i.e.*, if the pixel has not been written on and remains white, the neuron shall hold a 1.00 to represent white. As the levels of black increase in a pixel, the number stored in the neuron will approach 0.00. 0.00 will represent an entirely black pixel.

The 784 neurons containing the features of a 3 are the first layer in the network known as the IFM. At the output, the OFM of the network is a fully-connected layer and has ten neurons representing one of the digits zero (0) to nine (9). These neurons or operators that accept activation, weight and bias operands, and generate a number between 0.00 and 1.00 representing the probability of that digit being predicted from the IFM. Between the IFM and OFM is an arbitrary number of hidden layers, each layer containing an arbitrary number of neurons. The exact number of

hidden layers and hidden neurons are variables called hyperparameters and along with many other hyperparameters are used in tuning the model, and is the subject of significant research.

Each hidden layer is trained to detect more and more abstract representations of the image presented at the IFM. For example, as the first hidden layer of Figure B-1 might contain activations for all the curved segments of the loops of a digit, and the last hidden layer might contain activations for all the horizontal and vertical segments of the digit. The values of the activations a, $a_{N_i}^{L_{j-1}}$ of the neurons N in the preceding layer L, combined with a weighted w value $w_{N_i}^{L_{j-1}}$ of the connection, and some b bias b_{L_j}, determine the values of the activations of each neuron in a layer (see subsection B.2.5 for more about the training phase). The weights determine the *"strength"* of a connection, and the bias indicates how *"active"* the neuron tends to be. If there is a requirement for the weighted sum in any layer to be larger than the value one, *i.e.*, some bias is needed for inactivity, then bias is added to the weighted sum before it is passed to the activation function. The bias suggests how large the weighted sum of the neuron needs to be before the neuron becomes active.

The resultant weighted sum for a single neuron $a_{N_i}^{L_j}$ can is written as Equation 1:

$$a_{N_i}^{L_j} = (a_{N_i}^{L_{j-1}} \times w_{N_i}^{L_{j-1}} + a_{N_{i+1}}^{L_{j-1}} \times w_{N_{i+1}}^{L_{j-1}} + \ldots + a_{N_{i+n}}^{L_{j-1}} \times w_{N_{i+n}}^{L_{j-1}} + b_{L_j-1}) \tag{1}$$

where:

$a_{N_i}^{L_j}$ is the activation value (a) of neuron N_i in layer (L_j).

The bias is added to all neurons in the layer indicated as (b_{L_j-1}).

Activation Function

The non-linear activation function assists in determining if the output of the neuron should activate or *fire* based on a non-linear input [Jarrett et al. 2009]. Activation functions are often used between convolution layers and between the last layer and the fully connected layer. The activation value $a_{N_i}^{L_j}$ has a broad dynamic range; however, the resultant values are often required to be within a constrained range *e.g.*, 0 to 1. There are several functions, like sigmoid, tanh and ReLU that can compress the vast value range into a range between zero and one and a commonly used one for Perceptrons is the non-linear Logistic function or Sigmoid activation function (σ),

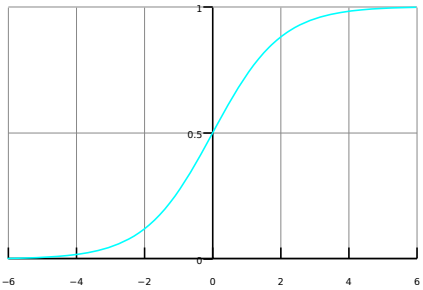


Figure B-2: **Logistic or Sigmoid Activation Function.**

see Equation 2 and Figure B-2. The sigmoid function trends large negative inputs to 0 and large positive numbers to 1. Therefore the sigmoid function gives a measure of how positive the weighted sum of the neuron is in Equation 1.

$$\sigma(a) = \frac{1}{1 + e^{-a}} \tag{2}$$

Gradients of sigmoids become infinitesimally small as the value of x increases, thus vanish, especially if the NN has many layers with multiple gradients that are approaching zero. Vanishing gradients negatively impact CNN training time as *"vanishing gradients make it difficult to know which direction the parameters should move to improve the cost function"* [Goodfellow et al. 2016]. Sigmoid likely always generates non-zero values and thus a dense representation. The activation function also allows stacking of multiple layers of neurons while retaining clear predictions. A more commonly used activation function today is the ReLU [Nair and Hinton 2010] due to its reduced likelihood of vanishing gradients [Glorot et al. 2011], as the gradient has a constant linear value for $a > 0$, which incidentally reduces computational complexity. ReLU also exhibits greater sparsity as all values are zeroed out when $a \leq 0$. ReLU tends to exhibit better convergence than Sigmoid [Krizhevsky et al. 2012]. However, ReLU is not without its issues. ReLU creates an unbounded activation as there is no means of constraining the output of the neuron as a increases. There is also the 'Dying ReLU' problem where the output tends to zero and thus 'dies', prohibiting learning and preventing convergence if there are many activations that are below zero. Many other types of activation can be used, such as Leaky ReLU, exponential linear unit (ELU), parametric rectified linear unit (PRELU), to name a few. Careful consideration of the activation function needs to be made when

designing a CNN, but ReLU suits the low-compute and low-storage requirements of low-power embedded systems.

Returning to the sigmoid function, the complete equation for one neuron of the multi-layer perceptron with sigmoid activation function is shown in Equation 3:

$$OFM_N = \sigma\left(\sum_{i=1}^{n} W_i a_i + b\right) \tag{3}$$

Every neuron of a layer is connected with a weighted connection and bias to every neuron in the next layer. Assume there are 49 arbitrarily chosen neurons in each of the two hidden layers, 784 input neurons from the 28×28 image presented at the IFM, then there are 784 weights per neuron and a bias for each of the neuron of each layer. This results in:

$$total\ weights = 784 \times 49 + 49 \times 49 + 49 \times 10 = 41,307 \tag{4}$$

$$total\ biases = 49 + 49 + 10 = 108 \tag{5}$$

The network contains a total of $41,415$ parameters (the addition of Equation 4 and Equation 5). Therefore, to predict the digit correctly, the network training has to converge on the correct combination of these $41,415$ weights and biases.

A more compact way to present the multipliers of Equation 1 required for every neuron in a layer is by using matrix-vector notation and perform matrix-vector arithmetic, see Equation 6.

$$OFM_{L_j} = \sigma\left(\begin{bmatrix} a_{N_i}^{L_j} \\ a_{N_{i+1}}^{L_j} \\ \vdots \\ a_{N_{i+n}}^{L_j} \end{bmatrix} \begin{bmatrix} w_{N_i}^{L_{j-1}} & w_{N_{i+1}}^{L_{j-1}} & \cdots & w_{N_{i+n}}^{L_{j-1}} \\ w_{N_i}^{L_j} & w_{N_{i+1}}^{L_j} & \cdots & w_{N_{i+n}}^{L_j} \\ \vdots & \vdots & \ddots & \vdots \\ w_{N_i}^{L_{j+k}} & w_{N_{i+1}}^{L_{j+k}} & \cdots & w_{N_{i+n}}^{L_{j+k}} \end{bmatrix} + \begin{bmatrix} b_{L_{j-1}} \\ b_{L_j} \\ \vdots \\ b_{L_{j+k}} \end{bmatrix}\right) \tag{6}$$

Equation 6 can be further simplified as Equation 7

$$OFM = \sigma\left(\mathbf{a}\mathbf{W}^T + \mathbf{b}\right) \tag{7}$$

where:

W^T represents the weights matrix;

a the activations;

b the biases.

Component representations of the neurons of the last hidden layer are combined and connected to the OFM. The neuron in the OFM layer with the highest value (top-1) shall be the predicted inferred digit, in this example, say, a 98% probability the network detected a 3 and, say, only a 20% probability the network suggested it was an 8.

The successive abstraction of the IFM into more and more fine-grained components for the OFM to predict are useful for different types of recognition from edge detection of items in an image to the breaking down of speech into phrases, sentences, words and letters.

Much the same as the multi-layer perceptron, the hidden neurons of the hidden layers are trained to detect certain features at different levels of abstraction within the image, *e.g.*, curves or straight lines, edges or not edges, eyes, nose, mouth, animal or not an animal, human or not!

There are many variants of DNNs used for the image recognition (CNNs) and time-variant streams such as speech recognition (LSTM), however, this chapter will focus on CNNs.

B.2.5 Training The Neural Network

Now that the essential inner workings of a feed-forward NN have been defined and how the NN recognises objects in a scene, the CNN algorithm needs to be trained for the recognition task.

The CNN is exposed to training data of, *e.g.*, 1000s of images of handwritten digits with text labels naming the digit. The network *learns* to recognise those digits with a desired high degree of prediction accuracy. The training stage initialises the CNN to a random value and runs a detection, that is a forward pass inference, to make a prediction. The prediction accuracy, no doubt low at first, is adjusted by propagating an error adjustment to the weights and biases in the CNN in a backward pass manner, called backpropagation. The error adjustment is calculated from

a cost function C_0, which is the square of the differences between the expected value of the output neuron and the actual value. In so far as the average of the cost functions over the entire training set tends to zero, the accurate prediction tends toward 100%.

We make another inference, and then test the prediction accuracy for any improvement. The weights are adjusted again in another backpropagation (backprop) based on the delta between the expected and actual prediction accuracy. The iterative process of inference and backprop continues until the accuracy of prediction of the input image converges on the desired accuracy, e.g. $95 - 98\%$ accuracy of predicting a 3 in the input image. To prevent over-fitting of prediction in the fully-connected layer, Krizhevsky *et al.*, proposed employing the dropout regularisation method [Krizhevsky et al. 2012].

Typically, the labelled training data is split up into three groups:

- a large training set (circa 70%) to train the CNN;
- a smaller validation set (circa 20%) to test the model for fit and prediction accuracy and tune the hyperparameters appropriately;
- a gold standard test set (circa 10%) that the CNN has not seen. The golden set tests how accurately the CNN classifies the new images.

Gold standard test sets are often used in competitions such as the ILSVRC and Kaggle competitions.

Neural Network Datasets

Datasets to train CNNs can be created and cleaned from scratch, which is extremely intensive work, or a predefined and cleaned dataset can be downloaded from many third-party sources. Some datasets are free and open-source while others may be closed and charge for their use. Google supplies an excellent search engine, [Google 2018] for finding the relevant datasets for the training task. For this simple multi-layer perceptron CNN LeCun *et al.*, [1998] produced a database of tens of thousands of labelled hand-written digits, called MNIST.

Stochastic Gradient Descent

The cost function essentially informs the weights and biases in the backpropagation phase how to change their values to improve the prediction accuracy.

From a calculus perspective, the cost function needs to find the input that minimises the value of the function, *i.e.*, the loss. However, for a complicated input such as the above CNN with $41,415$ parameters, there is a risk of finding only local minima without finding the global minimum, *i.e.*, the smallest value of the cost function. So, the weights and biases are updated with backpropagation (see subsection B.2.5 for more details). The global minimum is approached by iteratively making the step sizes and direction changes of the weights and biases proportional to the downward rate of change of slope and direction of the gradient, like a ball rolling down a hill. The negative sign of the step change required might indicate a downhill change of direction of weight value. The magnitude of the step change might represent the weight connections' impact on the cost function, thus which connections matter the most. Therefore, the gradient vector effectively encodes the relative importance of each weight and bias on the cost function. The nudging of the ball down the slope by taking gradually smaller negative deltas from the weights is called stochastic gradient descent (SGD);

- *stochastic* to reduce the compute requirements as the changes are made based on a randomised *"subset"* or *"batch"* of training data, *i.e.*, have a random probability distribution;
- *gradient descent* representing the change in weights and direction needed to head downhill.

The step sizes reduce as the slope reaches a horizontal level on the hill and thus, a local minimum. The gradual reduction in slope step size prevents overshooting the minimum (like the ball rolling to the bottom of a hill and back up the other side). However, changing the step size rate too quickly will increase the time to find the minimum, if ever (*e.g.*, the ball never settles at the bottom of the hill).

Backpropagation

Backpropagation is a method for computing and optimising the stochastic gradient descent (SGD). Briefly, the CNN is a huge differentiable cost function that takes data and weights at the input of a computational graph and produces a predicted loss at the output of the system.

SGD provides rapid reduction of the cost function and propagates that update to all the weights in the CNN. The SGD tries to highlight the impact of a weight value in the layer, so weights with more impact have stronger connections *i.e.*, larger values. If a digit 3 is the required output, the output layer weight corresponding to the digit 3 might be expected to be increased by a delta, and the rest of the weight values decreased by a delta. The deltas by which a weight is changed shall be proportional to how far from the target value the weight is at present. Backpropagation updates the bias b and weight W_i. The new activation a_i is proportional to the updated weight, W_i; SGD has the effect of strengthening the weights of the activations that detect a 3, and weakening the other weights.

Returning to the maths, let us simplify the multi-layer perceptron of Figure B-1 to a single neuron in each of the input, hidden and output layers. For this simple 4-layer-4-neuron multi-layer perceptron, if we assign a number label to each neuron from 1 for the input layer to 4 for the output layer, the cost function C becomes *C(w_1, b_1, w_2, b_2, w_3, b_3)*.

We need to discover the weight and bias values to which the cost function is most sensitive, *i.e.*, find values of W_L and b_L that results in the most efficient decrease in the cost function C.

The activations shall be labelled similarly to Equation 1. We label the output neuron a^L and the previously hidden layer a^{L-1} and continue along the layers, relabelling similarly.

If the required prediction y is 100% of predicting a 3, then the simple cost function C_0 is the square of the difference between the neurons activation value a^{L_j} and the desired value y, Equation 8

$$C_0 = (a^L - y)^2 \tag{8}$$

where C_0 is the cost function for all the weights and biases, the activation a^L, (see Equation 9), is a simplified version of Equation 1 for the activation of a single neuron in a layer.

$$a^L = \sigma(w^L a^{L-1} + b^L) \tag{9}$$

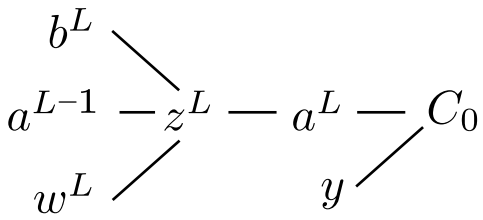


Figure B-3: **Cost Function Computational Graph.**

Let z^L equal to the non-sigmoid activation, Equation 10:

$$z^L = (w^L a^{L-1} + b^L) \tag{10}$$

Then the activation is simplified to Equation 11

$$a^L = \sigma(z^L) \tag{11}$$

The cost function computational graph of the Equation 8, Equation 10 and Equation 11, can be seen in Figure B-3.

We need to find the delta change in W^L needed to affect z^L. We also need to find how much that change in z^L affects a^{L-1} which, in turn with the desired y input, that delta change of a^L creates a delta change in the cost function C_0. Equation 12 shows these partial differential changes (∂) of the weights and is known as the Chain Rule and gives the sensitivity of C_0 to small changes in W^L.

$$\frac{\partial C_0}{\partial w^L} = \frac{\partial z^L}{\partial w^L}\frac{\partial a^L}{\partial z^L}\frac{\partial C0}{\partial a^L} \tag{12}$$

Now taking the derivative of Equation 8 gives cost function derivative, Equation 13:

$$\frac{\partial C_0}{\partial a^L} = 2(a^L - y) \tag{13}$$

Taking the derivative of Equation 9 gives the activation derivative Equation 14:

$$\frac{\partial a^L}{\partial z^L} = \sigma'(z^L) \tag{14}$$

And the derivative of the non-sigmoid activation, Equation 10 gives Equation 15

$$\frac{\partial z^L}{\partial w^L} = a^{L-1} \tag{15}$$

Bringing all the constituent parts of Equation 13, Equation 14 and Equation 15 together gives the cost function derivative Equation 16:

$$\begin{aligned} \frac{\partial C_0}{\partial w^L} &= \frac{\partial z^L}{\partial w^L}\frac{\partial a^L}{\partial z^L}\frac{\partial C_0}{\partial a^L} \\ &= a^{L-1}\sigma'(z^L)2(a^L - y) \end{aligned} \tag{16}$$

The cost function of Equation 16 is the function for one training example over one neuron, whereas the derivative of all neurons' cost functions, as an average of all the training examples, is needed, which can be seen in Equation 17.

$$\frac{\partial C}{\partial w^L} = \frac{1}{n}\sum_{k=0}^{n-1}\frac{\partial C_k}{\partial w^L} \tag{17}$$

The solution of Equation 17 is one component of the gradient vector, the sum of which is Equation 18.

$$\nabla C = \begin{bmatrix} \frac{\partial C}{\partial w^1} \\ \frac{\partial C}{\partial b^1} \\ \vdots \\ \frac{\partial C}{\partial w^L} \\ \frac{\partial C}{\partial b^L} \end{bmatrix} \tag{18}$$

As the sum of average derivative costs of weights has been found, a similar procedure is undertaken for the bias. This is easier as the framework of Equation 16 is used, substituting the weights' terms for the bias terms, *i.e.*, Equation 19.

$$\begin{aligned} \frac{\partial C_0}{\partial b^L} &= \frac{\partial z^L}{\partial b^L}\frac{\partial a^L}{\partial z^L}\frac{\partial C_0}{\partial a^L} \\ &= 1\sigma'(z^L)2(a^L - y) \end{aligned} \tag{19}$$

The process of Equation 8 to Equation 16 should be iteratively repeated for every layer preceding this until all the layers' weights and biases have been updated. Here the previous equations can be used as frameworks for the next layer, so taking

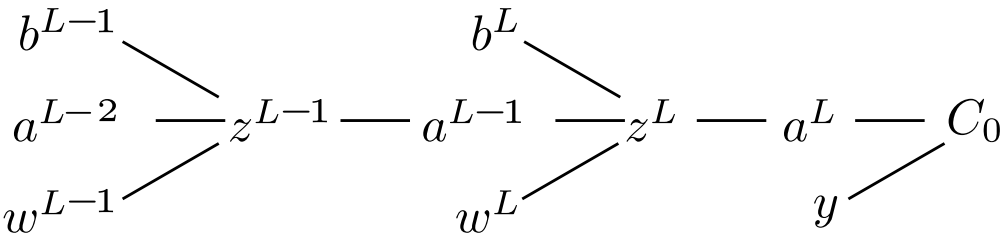


Figure B-4: **Cost Function Computational Graph for Multiple Layers.**

Equation 16, L can be substituted for $L-1$ and $L-1$ for $L-2$ and so on for each layer, as can be seen in Figure B-4.

The process above is repeated for all the neurons in all the layers. Firstly, the sum of all the cost functions of all the neurons in every layer is required, Equation 20.

$$C_0 = \sum_{j=0}^{n_{L-1}} (a_j^L - y_j)^2 \tag{20}$$

Now making z equal to the non-sigmoid activation for all layers and neurons, Equation 21:

$$z_j^L = w_{j0}^L a_0^{L-1} + \ldots + w_{j+n}^L a_{j+n}^{L-1} + b_j^L \tag{21}$$

The activation now becomes Equation 22:

$$a_j^L = \sigma(z_j^L) \tag{22}$$

The Chain rule derivative of the cost function for the weights of all the neurons in all layers becomes Equation 23:

$$\begin{aligned} \frac{\partial C_0}{\partial w_{jk}^L} &= \frac{\partial z_j^L}{\partial w_{jk}^L} \frac{\partial a_j^L}{\partial z_j^L} \frac{\partial C_0}{\partial a_j^L} \\ \frac{\partial C}{\partial w_{jk}^L} &= a_k^{L-1} \sigma'(z_j^L) \frac{\partial C}{\partial a_j^L} \end{aligned} \tag{23}$$

And the Chain Rule derivative of the cost function summed across all layers becomes Equation 24:

$$\begin{aligned}\frac{\partial C_0}{\partial a_k^{L-1}} &= \sum_{j=0}^{n_{L-1}} \frac{\partial z_j^L}{\partial a_k^{L-1}} \frac{\partial a_j^L}{\partial z_j^L} \frac{\partial C_0}{\partial a_j^L} \\ \frac{\partial C}{\partial a_{jk}^L} &= \sum_{j=0}^{n_{L+1}-1} w_{jk}^{L+1} \sigma'(z_j^{L+1}) \frac{\partial C}{\partial a_j^{L+1}} \\ &= 2(a_j^L - y_j)\end{aligned} \tag{24}$$

Therefore any particular neuron in the network influences the cost function through multiple weight connection paths and so all those influences have to be added to reach the final cost function for the entire network as shown by Equation 24.

The neural network model and training above introduced here demonstrates the computational complexity of the simplest neural networks. The network was configured to recognise a single digit in a single channel grey-scale image. The network works well if just numerical text is to be recognised, such as zip codes for the postal service. If more objects in colour images are to be recognised, then the above network needs to be further developed to produce a CNN.

B.3 Convolutional Neural Networks

CNNs build on the deep learning we've just encountered in subsection B.2.4. CNNs can recognise objects from multiple channels, usually red, green and blue of a colour photograph. CNNs are trained in much the same way as above with backpropagation and SGD. The major differences are that CNNs have scaled up the number of channels, layers and weights. With the previous artificial neural network model, it was supposed that the layers contain segments of a digit to be recognised, a similar assumption can be made here with the layers of weights containing aspects (or filters) of the image, again segments of the image such as curves or lines. Another significant difference CNNs exhibit compared to the *"Hello World"* of ANNs above is that CNNs do not have a weight per input pixel. The same weights are used across the channels of the IFM. CNNs are a little more complex again in that they allow us to detect and recognise much more complete patterns such as eyes, nose, mouth, person, man, woman, a dog with translation invariance characteristics, allowing the

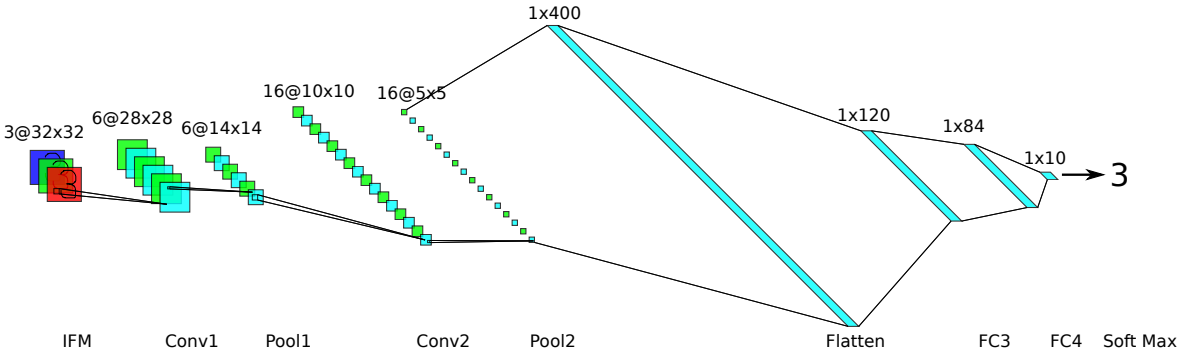


Figure B-5: **"Hello World" CNN Example.**

extraction of spatiotemporal features [Lecun et al. 1998]. Let us compare it with the *"Hello World"* of CNNs, LeNet-like network of Figure B-5.

The CNN takes an image (IFM) of 3-channel (red, green and blue) and dimensions 32×32 pixels and no padding of a digit 3 into the first layer. Six filters or kernels of dimensions 5×5 pixels slide across the image, from top left to bottom right, with a stride of 1, multiplying and accumulating the values with the corresponding image 5×5 tile and placing the results in the OFM layer ***Conv1***. Bias and a non-linearity such as sigmoid or ReLU[1]will be applied in ***Conv1***.

Before moving on, the detail of how the convolution operation alone works needs to be highlighted. Figure B-6 shows the CNN takes an IFM of C channels and dimensions $IH \times IW$ pixels[2]. M kernels are slid with a stride[3] of S across each of the C channels performing MAC operations. The MAC results is placed at the M channel point corresponding to the MAC operation.

Next in Figure B-5 is a ***Pooling*** layer which is applied to the OFM of the ***Conv1*** layer. In the Max Pooling layer, 2 filters and a stride of 2 are used. The max pool reduces the height and width of the representation by a factor of 2. The ***Pool1*** output of this pooling layer becomes 14×14 with the same 6 channels. Pool layers compresses height and width dimensions and not channel dimensions. The ***Conv1*** and ***Pool1*** complete the first layer of the CNN.

A further layer of Convolution of 16 kernels of dimensions 5×5 are applied to ***Pool1*** with a stride of 1 to produce a feature map ***Conv2*** of dimensions $10 \times 10 \times 16$.

[1]ReLU is more widely used as a non-linear activation function as it is less compute-intensive, requires less storage of resultant parameters and faster to converge during train.

[2]The image may also be padded with extra P pixels in width and height

[3]A stride of 1 means the kernel is moved up, down, left or right by 1-pixel after completion of the MAC operations.

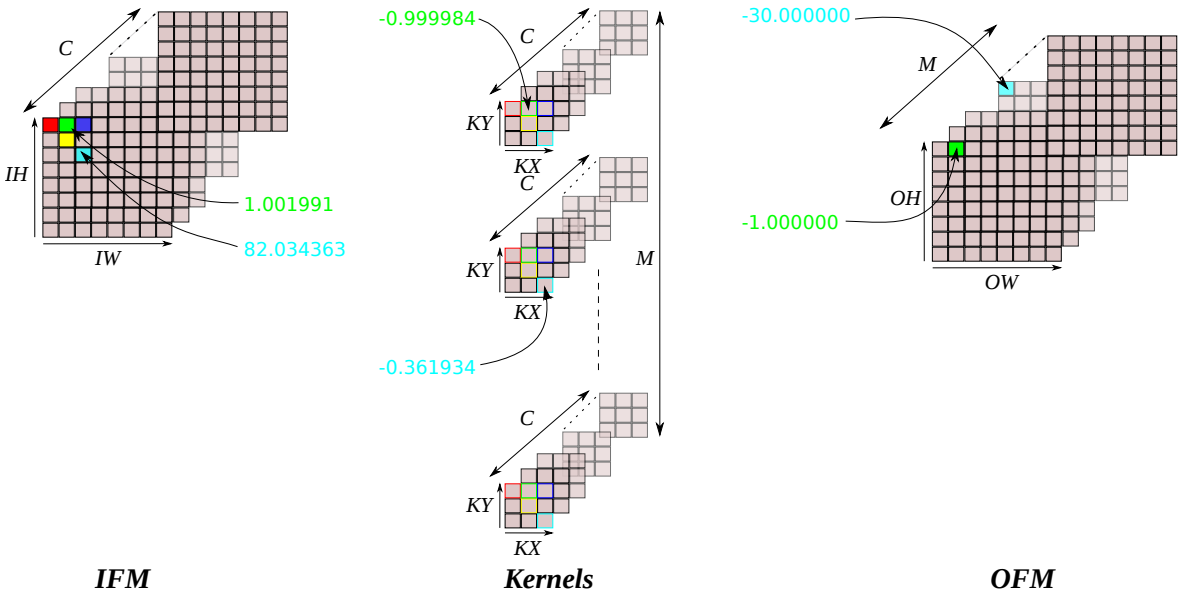


Figure B-6: **Naive Convolution Example.**

Max Pooling of 2 filters and stride of 2 are applied to ***Conv2*** to give ***Pool2*** of reduced height and width dimension and increased channels of $5 \times 5 \times 16$. ***Conv2*** and ***Pool2*** complete the second layer of the CNN.

Pool2 of Figure B-5 is now flattened or vectorised into a 400×1 vector of neurons. These 400 neurons are densely connected to the 120 neurons in a FC layer ***FC3***, reducing the vector to 120 neurons. The bias is also reduced to a vector of 120 values. A further FC layer is added with the bias to reduce the vector down to 84 values in ***FC4***. At the output classification layer, a softmax function is applied to ***FC4***. The softmax will have ten outputs corresponding to the 0 to 9 digits to be recognised by the CNN. The softmax of Figure B-5 will provide a probability of, say, 95% that the network recognised the 3 supplied at the CNN input.

B.3.1 Calculating the MAC Operations in a CNN

Up to 90% of the compute of a CNN is performed in the convolution layers [Farabet et al. 2010]. The performance of the compute within the convolution layer is mostly data movement and MAC operations. Optimising the data movement and MAC operations in the convolution layer would deliver efficient returns and potentially increase computational performance. Let us baseline the number of MACs operations that a naive convolution layer would perform. The convolution operation is determined by the following:

- Dimensions of the filter (or Kernel), $k \times k$;

Table B.1: Number of Arithmetic Operations, Activations and Parameters in CNNs.

	MAC	Comparisons	Addition	Division	Exponent	Activation	Parameters
VGG-16	154.7G	196.85M	10k	10k	10k	288.03M	138.36M
GoogleNet	16.04G	161.07M	8.83M	16.64M	8.83M	102.19M	7M
FCN-16s	89.38G	99.99M	5.26M	5.25M	5.25M	159.63M	134.82M
SqueezeNet v1.1	387.75M	6.02M	197k	1000	1000	7.69M	1.24M
ResNet-50	3.87G	10.89M	16.21M	10.59M	1000	46.72M	25.56M
ResNet-152	11.3G	22.33M	35.27M	22.03M	1000	100.11M	60.19M
YOLO	20.29G	21.83M	0	0	0	30.22M	271.7M
ZynqNet	529.3M	3.22M	66.56k	1.02k	1.02k	8.61M	2.53M
Inception V4	12.27G	21.87M	53.42M	15.09M	1000	72.56M	42.71M

- Number of filters in the layer, M;
- Dimensions of the IFM, $H \times W$;
- Number of batches of IFM, N;
- Number of IFM channels, C;
- Stride of the kernel over the IFM, S.

Using these parameters, the maximum number of MAC operations in a convolution stage of a CNN can be calculated as Equation 25.

$$Number\ of\ MACs = \frac{H \times W \times C \times M \times k \times k \times N}{S^2} \tag{25}$$

Taking VGG-16's IFM first layer as an example, its dimensions are:

- $k = 3$;
- $M = 64$;
- $H = W = 224$;
- $N = 10$;
- $C = 3$;
- $S = 1$.

So the number of MACs in the first convolution layer are Equation 26. The calculation for a single layer demonstrates the very large number of MACs, compounded with multiple convolution layers, confirmed by Table B.1.

$$\begin{aligned} Number\ of\ MACs &= \frac{224 \times 224 \times 3 \times 64 \times 3 \times 3 \times 10}{1^2} \\ &= 867,041,280 \end{aligned} \tag{26}$$

Bibliography

Abadi, Martín et al. (2016). "Tensorflow: A system for large-scale machine learning". In: *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pp. 265–283.

ABB (1974). *ABB IRB 6 Robot*.

Ackley, David H et al. (1985). "A learning algorithm for Boltzmann machines". In: *Cognitive science* 9.1, pp. 147–169.

Akkaya, Ilge et al. (2019). "Solving Rubik's Cube with a Robot Hand". In: *arXiv preprint arXiv:1910.07113*.

Alipanahi, Babak et al. (2015). "Predicting the sequence specificities of DNA-and RNA-binding proteins by deep learning". In: *Nature biotechnology* 33.8, p. 831.

Amdahl, Gene M. (1967). *Validity of the single processor approach to achieving large scale computing capabilities*.

Anderson, Andrew et al. (2017). "Efficient Multibyte Floating Point Data Formats Using Vectorization". In: *IEEE Transactions on Computers* 66.12, pp. 2081–2096.

Anderson, Andrew et al. (2019). "Hardware and software performance in deep learning". In: *Many-Core Computing: Hardware and Software*. Ed. by Geoff V. Merrett Bashir M. Al-Hashimi. Computing. Institution of Engineering and Technology. Chap. 6, pp. 141–161.

Arm (2019). *Arm Neon Intrinsics Reference for ACLE Q2 2019*.

Arulkumaran, Kai et al. (2019). "Alphastar: An evolutionary computation perspective". In: *arXiv preprint arXiv:1902.01724*.

Azevedo, Frederico AC et al. (2009). "Equal numbers of neuronal and nonneuronal cells make the human brain an isometrically scaled-up primate brain". In: *Journal of Comparative Neurology* 513.5, pp. 532–541.

Baker, Bowen et al. (2019). "Emergent tool use from multi-agent autocurricula". In: *arXiv preprint arXiv:1909.07528*.

Bambha, Manik et al. (2011). *Automatic Image Tagging*.

Bao, Jianmin et al. (2017). "CVAE-GAN: fine-grained image generation through asymmetric training". In: *Proceedings of the IEEE international conference on computer vision*, pp. 2745–2754.

Boser, Bernhard E et al. (1992). "A training algorithm for optimal margin classifiers". In: *Proceedings of the fifth annual workshop on Computational learning theory*. ACM, pp. 144–152.

Brayton, Robert and Alan Mishchenko (2010). "ABC: An Academic Industrial-Strength Verification Tool". In: Computer Aided Verification. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 24–40.

Brown, Adam (2016). *Adaptive algorithms for interrogating the viewable scene of an automotive radar*.

Bryant, William Cullen (1870a). *The Iliad of Homer: translated into English blank verse*. Houghton, Mifflin and Company.

— (1870b). *The Iliad of Homer: translated into English blank verse*. Houghton, Mifflin and Company.

— (1870c). *The Iliad of Homer: translated into English blank verse*. Houghton, Mifflin and Company.

Bucilǎu, Cristian et al. (2006). *Model compression*.

Bush, Vannevar and Vannevar Bush (1945). "As we may think". In: *Resonance* 5.11.

Cambria, Erik and Bebo White (2014). "Jumping NLP curves: A review of natural language processing research". In: *IEEE Computational intelligence magazine* 9.2, pp. 48–57.

Capek, Karel (2004). *RUR (Rossum's universal robots)*. Penguin.

Chen, Chenyi et al. (2015a). "Deepdriving: Learning affordance for direct perception in autonomous driving". In: *Proceedings of the IEEE International Conference on Computer Vision*, pp. 2722–2730.

Chen, Tianshi et al. (2014). "DianNao: a small-footprint high-throughput accelerator for ubiquitous machine-learning". In: *ACM SIGPLAN Notices* 49.4, pp. 269–284.

Chen, Y. et al. (2016). "Eyeriss: A Spatial Architecture for Energy-Efficient Dataflow for Convolutional Neural Networks". In: *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, pp. 367–379.

Chen, Yu Hsin et al. (2017). "Eyeriss: An Energy-Efficient Reconfigurable Accelerator for Deep Convolutional Neural Networks". In: *IEEE Journal of Solid-State Circuits* 52.1.

Chen, Yu-Hsin et al. (2019). "Eyeriss v2: A flexible accelerator for emerging deep neural networks on mobile devices". In: *IEEE Journal on Emerging and Selected Topics in Circuits and Systems* 9.2, pp. 292–308.

Chen, Yunji et al. (2015b). "DaDianNao: A Machine-Learning Supercomputer". In: *Proceedings of the Annual International Symposium on Microarchitecture, MICRO*.

Cho, Minsik and Daniel Brand (2017). "MEC: memory-efficient convolution for deep neural network". In: *Proceedings of the 34th International Conference on Machine Learning-Volume 70*. JMLR. org, pp. 815–824.

Chollet, François (2015). *Keras*.

Choquette, Jack et al. (2021). "NVIDIA A100 Tensor Core GPU: Performance and Innovation". In: *IEEE Micro* 01.01, pp. 1–1.

Chung, E. et al. (2018). "Serving DNNs in Real Time at Datacenter Scale with Project Brainwave". In: *IEEE Micro* 38.2, pp. 8–20.

Church, Alonzo (1937). "AM Turing. On computable numbers, with an application to the Entscheidungs problcm. Proceedings of the London Mathematical Society, 2 s. vol. 42 (1936-1937), pp. 230-265". In: *The Journal of Symbolic Logic* 2.1, pp. 42–43.

Collobert, Ronan et al. (2011). "Natural language processing (almost) from scratch". In: *Journal of machine learning research* 12.Aug, pp. 2493–2537.

Cook, Steven J et al. (2019). "Whole-animal connectomes of both Caenorhabditis elegans sexes". In: *Nature* 571.7763, pp. 63–71.

Cortes, Corinna and Vladimir Vapnik (1995). "Support-vector networks". In: *Machine Learning* 20.3, pp. 273–297.

Courbariaux, Matthieu et al. (2015a). “BinaryConnect: Training Deep Neural Networks with binary weights during propagations”. In:

— (2015b). *Low precision arithmetic for deep learning*.

Courbariaux, Matthieu et al. (2016). “Binarized Neural Networks: Training Deep Neural Networks with Weights and Activations Constrained to +1 or -1”. In:

Dahl, G. E. et al. (2013). “Improving deep neural networks for LVCSR using rectified linear units and dropout”. In: *2013 IEEE International Conference on Acoustics, Speech and Signal Processing*, pp. 8609–8613.

DARPA (2015). *DARPA Robotics Challenge (DRC) (Archived)*.

De Dinechin, F. et al. (2009). “Generating high-performance custom floating-point pipelines”. In: *2009 International Conference on Field Programmable Logic and Applications*. USA: IEEE, pp. 59–64.

De Dinechin, Florent and Bogdan Pasca (2011). “Designing custom arithmetic data paths with FloPoCo”. In: *IEEE Design and Test of Computers* 28.4, pp. 18–27.

Delbruck, Tobi (2008). “Frame-free dynamic digital vision”. In: *Proceedings of Intl. Symp. on Secure-Life Electronics, Advanced Electronics for Quality Life and Society*, pp. 21–26.

Deng, Jia et al. (2009). “Imagenet: A large-scale hierarchical image database”. In: *2009 IEEE conference on computer vision and pattern recognition*. Ieee, pp. 248–255.

Deng, Li et al. (2013). “Recent advances in deep learning for speech research at Microsoft”. In: *2013 IEEE International Conference on Acoustics, Speech and Signal Processing*. IEEE, pp. 8604–8608.

Dennard, Robert H et al. (1974). “Design of ion-implanted MOSFET’s with very small physical dimensions”. In: *IEEE Journal of Solid-State Circuits* 9.5, pp. 256–268.

Dettmers, Tim (2015). “8-Bit Approximations for Parallelism in Deep Learning”. In: *International Conference On Learning Representations* 2, pp. 1–9.

Devlin, Jacob et al. (2018). “Bert: Pre-training of deep bidirectional transformers for language understanding”. In: *arXiv preprint arXiv:1810.04805*.

Devol, Jr George C (1961). “Programmed article transfer”. Pat.

DiCecco, Roberto et al. (2017). "FPGA-based training of convolutional neural networks with a reduced precision floating-point library". In: *2017 International Conference on Field Programmable Technology (ICFPT)*. IEEE, pp. 239–242.

Dikmen, Murat and Catherine M. Burns (2016). *Autonomous Driving in the Real World: Experiences with Tesla Autopilot and Summon*.

Dinelli, Gianmarco et al. (2020). "Advantages and limitations of fully on-chip CNN FPGA-based hardware accelerator". In: *2020 IEEE International Symposium on Circuits and Systems (ISCAS)*. IEEE, pp. 1–5.

Dynamics, Boston (2019). *Boston Dynamics: Changing your idea of what robots can do*.

Esteva, Andre et al. (2017). "Dermatologist-level classification of skin cancer with deep neural networks". In: *Nature* 542.7639, p. 115.

Fan, Deng-Ping et al. (2020). "Inf-Net: Automatic COVID-19 Lung Infection Segmentation from CT Images". In: *IEEE Transactions on Medical Imaging*.

Farabet, C. et al. (2010). "Hardware accelerated convolutional neural networks for synthetic vision systems". In: *Proceedings of 2010 IEEE International Symposium on Circuits and Systems*. Paris, France: IEEE ISCAS, pp. 257–260.

Farabet, C. et al. (2011). "NeuFlow: A runtime reconfigurable dataflow processor for vision". In: *CVPR 2011 WORKSHOPS*, pp. 109–116.

Faraone, Christopher A (1987). "Hephaestus the magician and Near Eastern parallels for Alcinous' watchdogs". In: *Greek, Roman, and Byzantine Studies* 28.3, pp. 257–280.

Ferrucci, David et al. (2013). "Watson: beyond jeopardy!" In: *Artificial Intelligence* 199, pp. 93–105.

Fisher, Randall J and Henry G Dietz (1998). "Compiling for SIMD within a register". In: *International Workshop on Languages and Compilers for Parallel Computing*. Berlin, Heidelberg: Springer, pp. 290–305.

Flamand, Eric et al. (2018). "GAP-8: A RISC-V SoC for AI at the Edge of the IoT". In: *2018 IEEE 29th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*. IEEE, pp. 1–4.

Fowers, Jeremy et al. (2018). "A configurable cloud-scale DNN processor for real-time AI". In: *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, pp. 1–14.

Frossard, Davi (2016). "Macroarchitecture of VGG16". In: Block Diagram of the macro architecture of VGG16.

Fu, Yao et al. (2016). "Deep Learning with INT8 Optimization on Xilinx Devices White Paper (WP485)". In: 486.WP486 (v1.0.1), pp. 1–11.

Fukushima, Kunihiko (1980). "Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position". In: *Biological Cybernetics* 36.4, pp. 193–202.

Fürer, Martin (2007). "Faster integer multiplication". In: San Diego, California, USA: ACM, pp. 57–66.

Gangadharan, Sridhar and Sanjay Churiwala (2013). *Constraining designs for synthesis and timing analysis : a practical guide to synopsys design constraints (SDC)*. Springer.

Garland, James and David Gregg (2017). "Low Complexity Multiply Accumulate Unit for Weight-Sharing Convolutional Neural Networks". In: *IEEE Computer Architecture Letters* 16.2, pp. 132–135. ISSN: 1556-6056.

— (2018). "Low Complexity Multiply-Accumulate Units for Convolutional Neural Networks with Weight-Sharing". In: *ACM Transactions on Architecture and Code Optimization* 15.3, 31:1–31:24. ISSN: 1544-3566.

— (2021). "HOBFLOPS for CNNs: Hardware Optimized Bitslice-Parallel Floating-Point Operations for Convolutional Neural Networks". In: *PREPRINT (Version 1) available at Research Square*.

Girshick, Ross (2015). "Fast R-CNN". In: *2015 IEEE International Conference on Computer Vision (ICCV)*, pp. 1440–1448.

Girshick, Ross et al. (2014a). "Rich feature hierarchies for accurate object detection and semantic segmentation". In: *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 580–587.

— (2014b). "Rich feature hierarchies for accurate object detection and semantic segmentation". In: *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 580–587.

Glorot, Xavier et al. (2011). "Deep sparse rectifier neural networks". In: *Proceedings of the fourteenth international conference on artificial intelligence and statistics*, pp. 315–323.

Gong, Yunchao et al. (2014). "Compressing deep convolutional networks using vector quantization". In: *arXiv preprint arXiv:1412.6115*.

Goodfellow, Ian et al. (2014). "Generative adversarial nets". In: *Advances in neural information processing systems*, pp. 2672–2680.

Goodfellow, Ian et al. (2016). *Deep Learning*. MIT Press.

Google (2018). *Google Dataset Search*.

— (2019). *BFloat16: The secret to high performance on Cloud TPUs*.

Gupta, Saurabh et al. (2017). "Cognitive mapping and planning for visual navigation". In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 2616–2625.

Gupta, Suyog et al. (2015). "Deep Learning with Limited Numerical Precision". In: *Proceedings of the 32nd International Conference on International Conference on Machine Learning - Volume 37*. JMLR.org.

Han, Song et al. (2016b). "EIE: Efficient Inference Engine on Compressed Deep Neural Network". In: *Proceedings - 2016 43rd International Symposium on Computer Architecture, ISCA 2016*, pp. 243–254.

Han, Song et al. (2016a). "Deep Compression: Compressing Deep Neural Network with Pruning, Trained Quantization and Huffman Coding". In: *CoRR* abs/1510.00149.

Hauser, John (1999). "The SoftFloat and TestFloat Validation Suite for Binary Floating-Point Arithmetic". In: *University of California, Berkeley, Tech. Rep*.

He, Kaiming et al. (2015). "Deep Residual Learning for Image Recognition". In: *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*.

Hershey, Shawn et al. (2017). "CNN architectures for large-scale audio classification". In: *2017 ieee international conference on acoustics, speech and signal processing (icassp)*. IEEE, pp. 131–135.

Hillis, W Daniel (1989). *The connection machine*. MIT press.

Hinton, Geoffrey et al. (2012). "Deep neural networks for acoustic modeling in speech recognition". In: *IEEE Signal processing magazine* 29.

Hinton, Geoffrey et al. (2015). "Distilling the knowledge in a neural network". In: *arXiv preprint arXiv:1503.02531*.

Hinton, Geoffrey E. (1989). "Connectionist learning procedures". In: *Artificial Intelligence* 40.1, pp. 185–234.

Hinton, Geoffrey E. (2002). "Training products of experts by minimizing contrastive divergence". In: *Neural Comput.* 14.8, pp. 1771–1800.

Hinton, Geoffrey E and Ruslan R Salakhutdinov (2006). "Reducing the dimensionality of data with neural networks". In: *science* 313.5786, pp. 504–507.

Hinton, Geoffrey E et al. (1984). *Boltzmann machines: Constraint satisfaction networks that learn*. Carnegie-Mellon University, Department of Computer Science Pittsburgh.

Hinton, Geoffrey E. et al. (2006). "A fast learning algorithm for deep belief nets". In: *Neural Comput.* 18.7, pp. 1527–1554.

Hochreiter, Sepp and Jürgen Schmidhuber (1997). "Long short-term memory". In: *Neural computation* 9.8, pp. 1735–1780.

Holland, S. W. et al. (1979). "CONSIGHT-I: A Vision-Controlled Robot System for Transferring Parts from Belt Conveyors". In: *Computer Vision and Sensor-Based Robots*. Ed. by George G. Dodd and Lothar Rossol. Boston, MA: Springer US, pp. 81–100.

Hopfield, J J (1982). "Neural networks and physical systems with emergent collective computational abilities". In: *Proceedings of the National Academy of Sciences* 79.8, pp. 2554–2558.

Howard, Andrew G et al. (2017). "Mobilenets: Efficient convolutional neural networks for mobile vision applications". In: *arXiv preprint arXiv:1704.04861*.

Hu, Jie et al. (2018). "Squeeze-and-excitation networks". In: *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 7132–7141.

Hubel, David H and Torsten N Wiesel (1959). "Receptive fields of single neurones in the cat's striate cortex". In: *The Journal of physiology* 148.3, pp. 574–591.

Iandola, Forrest N et al. (2016). *SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and< 0.5 MB model size*.

IEEE (2019). "IEEE Standard for Floating-Point Arithmetic". In: *IEEE Std 754-2019 (Revision of IEEE 754-2008)*, pp. 1–84.

Intel. *Intel Agrees to Acquire Nervana Systems*.

— *Intel Completes Acquisition of Altera*.

— *Intel to Acquire Artificial Intelligence Chipmaker Habana Labs*.

— *Intel to Acquire Movidius*.

— (2020). *Intel Intrinsics Guide*.

Ionica, Mircea Horea and David Gregg (2015). "The movidius myriad architecture's potential for scientific computing". In: *IEEE Micro* 35.1, pp. 6–14.

Jarrett, Kevin et al. (2009). "What is the best multi-stage architecture for object recognition?" In: *2009 IEEE 12th international conference on computer vision*. IEEE, pp. 2146–2153.

Jermyn, Michael et al. (2016). "Neural networks improve brain cancer detection with Raman spectroscopy in the presence of operating room light artifacts". In: *Journal of biomedical optics* 21.9, p. 094002.

Johnson, Jeff (2018). "Rethinking floating point for deep learning". In: *arXiv preprint arXiv:1811.01721*.

Jouppi, Norman P. et al. (2017). "In-Datacenter Performance Analysis of a Tensor Processing Unit". In: *SIGARCH Comput. Archit. News* 45.2, pp. 1–12. ISSN: 0163-5964.

Kang, Hyeong-Ju (2018). "Short floating-point representation for convolutional neural network inference". In: *IEICE Electronics Express*, p. 15.20180909.

Kang, Minsoo (2002). "Wonders of Mathematical Magic: Lists of Automata in the Transition from Magic to Science, 1533-1662". In: *Comitatus: A Journal of Medieval and Renaissance Studies* 33.1, pp. 113–139.

Kapil, Ansh et al. (2018). "Deep semi supervised generative learning for automated tumor proportion scoring on NSCLC tissue needle biopsies". In: *Scientific reports* 8.1, pp. 1–10.

Kasparov, Garry and Mig Greengard (2018). *Deep Thinking: Where Machine Intelligence Ends and Human Creativity Begins*. Perseus Books.

Khosla, Aditya et al. (2011). "Novel dataset for fine-grained image categorization: Stanford dogs". In: *Proc. CVPR Workshop on Fine-Grained Visual Categorization (FGVC)*. Vol. 2.

Kim, Hyongsuk and Shyam Prasad Adhikari (2012). "Memistor is not memristor [express letters]". In: *IEEE Circuits and Systems Magazine* 12.1, pp. 75–78.

Koren, Yehuda (2009). "The bellkor solution to the netflix grand prize". In: *Netflix prize documentation* 81.2009, pp. 1–10.

Krizhevsky, Alex and Geoffrey Hinton (2009). *Learning multiple layers of features from tiny images*. Tech. rep. Citeseer.

Krizhevsky, Alex et al. (2012). "ImageNet Classification with Deep Convolutional Neural Networks". In: *Advances In Neural Information Processing Systems*, pp. 1–9.

Lample, Guillaume et al. (2017). "Unsupervised machine translation using monolingual corpora only". In: *arXiv preprint arXiv:1711.00043*.

LeCun, Y. et al. (1989). "Backpropagation Applied to Handwritten Zip Code Recognition". In: *Neural Computation* 1.4, pp. 541–551.

Lecun, Y. et al. (1998). "Gradient-based learning applied to document recognition". In: *Proceedings of the IEEE* 86.11, pp. 2278–2324.

LeCun, Yann et al. (1998). "The MNIST database of handwritten digits, 1998". In: *URL http://yann. lecun. com/exdb/mnist* 10, p. 34.

LeCun, Yann et al. (1999). "Object recognition with gradient-based learning". In: *Shape, contour and grouping in computer vision*. Springer, pp. 319–345.

LeCun, Yann et al. (2015). "Deep learning". In: *Nature* 521.7553, pp. 436–444.

Lettvin, J. Y. et al. (1959). "What the Frog's Eye Tells the Frog's Brain". In: *Proceedings of the IRE* 47.11, pp. 1940–1951.

Levine, Sergey et al. (2016). "End-to-end training of deep visuomotor policies". In: *The Journal of Machine Learning Research* 17.1, pp. 1334–1373.

Li, Bo-hu et al. (2017a). "Applications of artificial intelligence in intelligent manufacturing: a review". In: *Frontiers of Information Technology & Electronic Engineering* 18.1, pp. 86–96.

Li, Fei-Fei et al. (2017b). *Stanford CS class CS231n: Convolutional Neural Networks for Visual Recognition*.

Lin, Darryl et al. (2016). "Fixed point quantization of deep convolutional networks". In: *International Conference on Machine Learning*, pp. 2849–2858.

Liu, Bo et al. (2019). "An Ultra-Low Power Always-On Keyword Spotting Accelerator Using Quantized Convolutional Neural Network and Voltage-Domain Analog Switching Network-Based Approximate Computing". In: *IEEE Access* 7, pp. 186456–186469.

Lo, Chun Y et al. (2018). "Fixed-point implementation of convolutional neural networks for image classification". In: *2018 International Conference on Advanced Technologies for Communications (ATC)*. IEEE, pp. 105–109.

Long, Jonathan et al. (2015). "Fully convolutional networks for semantic segmentation". In: *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 3431–3440.

Ma, Yufei et al. (2017). "Optimizing Loop Operation and Dataflow in FPGA Acceleration of Deep Convolutional Neural Networks". In: *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays - FPGA '17*, pp. 45–54.

Maghdid, Halgurd S et al. (2020). "A novel ai-enabled framework to diagnose coronavirus covid 19 using smartphone embedded sensors: Design study". In: *arXiv preprint arXiv:2003.07434*.

Mao, H. et al. (2017). "Exploring the Granularity of Sparsity in Convolutional Neural Networks". In: *2017 IEEE Conference on Computer Vision and Pattern Recognition Workshops (CVPRW)*, pp. 1927–1934.

Mathur, Maya B and David B Reichling (2016). "Navigating a social world with robot partners: A quantitative cartography of the Uncanny Valley". In: *Cognition* 146, pp. 22–32.

McCarthy, John et al. (2006). "A proposal for the dartmouth summer research project on artificial intelligence, august 31, 1955". In: *AI magazine* 27.4, pp. 12–12.

McCulloch, Warren S and Walter Pitts (1943). "A logical calculus of the ideas immanent in nervous activity". In: *The bulletin of mathematical biophysics* 5.4, pp. 115–133.

Mead, Carver (1989). "Analog VLSI and neutral systems". In: *NASA STI/Recon Technical Report A* 90.

Minsky, Marvin and Seymour A Papert (1969). *Perceptrons: An introduction to computational geometry*. MIT press.

Mittal, Sparsh (2016). "A survey of techniques for approximate computing". In: *ACM Computing Surveys (CSUR)* 48.4, pp. 1–33.

Moloney, David et al. (2014). "Myriad 2: Eye of the computational vision storm". In: *2014 IEEE Hot Chips 26 Symposium (HCS)*. IEEE, pp. 1–18.

Moons, Bert and Marian Verhelst (2016). "A 0.3–2.6 TOPS/W precision-scalable processor for real-time large-scale ConvNets". In: *2016 IEEE Symposium on VLSI Circuits (VLSI-Circuits)*. IEEE, pp. 1–2.

Moons, Bert et al. (2016). "Energy-efficient convnets through approximate computing". In: *2016 IEEE Winter Conference on Applications of Computer Vision (WACV)*. IEEE, pp. 1–8.

Moore, Gordon E (1965). *Cramming more components onto integrated circuits*.

Movidius, Intel (2017). *Intel Movidius Myriad 2 VPU Enables Advanced Computer Vision and Deep Learning Features in Ultra-Compact DJI Spark Drone*.

Nair, Vinod and Geoffrey E Hinton (2010). "Rectified linear units improve restricted boltzmann machines". In: *ICML*.

Netzer, Yuval et al. (2011). "Reading digits in natural images with unsupervised feature learning". In:

Ning, Yishuang et al. (2019). "A Review of Deep Learning Based Speech Synthesis". In: *Applied Sciences* 9.19.

NVidia (2020). *NVIDIA A100 Tensor Core GPU Architecture*.

Paszke, Adam et al. (2017). "Pytorch: Tensors and dynamic neural networks in python with strong gpu acceleration". In: *PyTorch: Tensors and dynamic neural networks in Python with strong GPU acceleration* 6.

Pfeiffer, Mark et al. (2017). "From perception to decision: A data-driven approach to end-to-end motion planning for autonomous ground robots". In: *2017 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE, pp. 1527–1533.

Pham, Phi-Hung et al. (2012). "NeuFlow: Dataflow vision processing system-on-a-chip". In: *2012 IEEE 55th International Midwest Symposium on Circuits and Systems (MWSCAS)*. IEEE, pp. 1044–1047.

Quigley, Morgan et al. (2009). "ROS: an open-source Robot Operating System". In: *ICRA workshop on open source software*. Vol. 3. Kobe, Japan, p. 5.

Raibert, Marc (2017). *Meet Spot, the robot dog that can run, hot and open doors*.

Rastegari, Mohammad et al. (2016). "Xnor-net: Imagenet classification using binary convolutional neural networks". In: *European Conference on Computer Vision*. Springer, pp. 525–542.

Redmon, Joseph et al. (2016). "You only look once: Unified, real-time object detection". In: *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 779–788.

Roland, Alex et al. (2002). *Strategic computing: DARPA and the quest for machine intelligence, 1983-1993*. MIT Press.

Rosenblatt, Frank (1958). "The perceptron: a probabilistic model for information storage and organization in the brain". In: *Psychological review* 65.6, p. 386.

— (1961). *Principles of neurodynamics. perceptrons and the theory of brain mechanisms*. Tech. rep. Cornell Aeronautical Lab Inc Buffalo NY.

Rumelhart, David E et al. (1985). *Learning internal representations by error propagation*. Tech. rep. California Univ San Diego La Jolla Inst for Cognitive Science.

Russakovsky, Olga et al. (2015). "Imagenet large scale visual recognition challenge". In: *International journal of computer vision* 115.3, pp. 211–252.

Ryu, Sungju et al. (2018). "Feedforward-Cutset-Free Pipelined Multiply–Accumulate Unit for the Machine Learning Accelerator". In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 27.1, pp. 138–146.

Rzayev, Tayyar et al. (2017). "DeepRecon: Dynamically reconfigurable architecture for accelerating deep neural networks". In: *2017 International Joint Conference on Neural Networks (IJCNN)*. IEEE, pp. 116–124.

Sabeetha, S. et al. (2015). "A study of performance comparison of digital multipliers using 22nm strained silicon technology". In: *2015 2nd International Conference on Electronics and Communication Systems (ICECS)*, pp. 180–184.

Sabour, Sara et al. (2017). "Dynamic routing between capsules". In: *Advances in neural information processing systems*, pp. 3856–3866.

Samuel, Arthur L (1959). "Some studies in machine learning using the game of checkers". In: *IBM Journal of research and development* 3.3, pp. 210–229.

Schuster, Mike and Kuldip K Paliwal (1997). "Bidirectional recurrent neural networks". In: *IEEE Transactions on Signal Processing* 45.11, pp. 2673–2681.

Schwartz, Roy et al. (2019). "Green AI". In: *arXiv preprint arXiv:1907.10597*.

Seshadri, Vivek et al. (2017). "Ambit: In-memory accelerator for bulk bitwise operations using commodity DRAM technology". In: *2017 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, pp. 273–287.

Shalev-Shwartz, Shai et al. (2016). "Safe, multi-agent, reinforcement learning for autonomous driving". In: *arXiv preprint arXiv:1610.03295*.

Shannon, Claude Elwood (1948). "A mathematical theory of communication". In: *Bell system technical journal* 27.3, pp. 379–423.

Sharma, Sagar (2017). *What the Hell is Perceptron?*

Silver, David et al. (2017). "Mastering the game of Go without human knowledge". In: *Nature* 550.7676, pp. 354–359.

Simonyan, Karen and Andrew Zisserman (2014a). "Two-stream convolutional networks for action recognition in videos". In: *Advances in neural information processing systems*, pp. 568–576.

— (2014b). *Very Deep Convolutional Networks for Large-Scale Image Recognition*.

Srivastava, Nitish (2013). "Improving neural networks with dropout". In: *University of Toronto* 182, p. 566.

Sutton, Richard S (1988). "Learning to predict by the methods of temporal differences". In: *Machine learning* 3.1, pp. 9–44.

Suwajanakorn, Supasorn et al. (2017). "Synthesizing Obama: learning lip sync from audio". In: *ACM Transactions on Graphics* 36.4, pp. 1–13.

Sze, Vivienne et al. (2017). "Efficient processing of deep neural networks: A tutorial and survey". In: *Proceedings of the IEEE* 105.12, pp. 2295–2329.

Szegedy, C. et al. (2015). "Going deeper with convolutions". In: *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. Vol. 07-12-June, pp. 1–9.

Tagliavini, Giuseppe et al. (2018). "Flexfloat: A software library for transprecision computing". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 39.1, pp. 145–156.

Taigman, Yaniv et al. (2014). "Deepface: Closing the gap to human-level performance in face verification". In: *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 1701–1708.

Team, Theano Development et al. (2016). "Theano: A Python framework for fast computation of mathematical expressions". In: *arXiv preprint arXiv:1605.02688*.

Thies, J. et al. (2016). "Face2Face: Real-Time Face Capture and Reenactment of RGB Videos". In: *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 2387–2395.

Thorpe, Charles et al. (1988). "Vision and navigation for the Carnegie-Mellon Navlab". In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 10.3, pp. 362–373.

Thrun, S and M Montemerlo (2005). *DARPA Grand Challenge 2005 Technical Paper*.

Touvron, Hugo et al. (2020). "Fixing the train-test resolution discrepancy: FixEfficientNet". In: *arXiv preprint arXiv:2003.08237*.

Tung, C. and S. Huang (2020). "A High-Performance Multiply-Accumulate Unit by Integrating Additions and Accumulations Into Partial Product Reduction Process". In: *IEEE Access* 8, pp. 87367–87377.

Turing, A. M. (1950). "Computing Machinery and Intelligence". In: *Mind* LIX.236, pp. 433–460.

Umuroglu, Yaman et al. (2016). "FINN: A Framework for Fast, Scalable Binarized Neural Network Inference". In:

Umuroglu, Yaman et al. (2017). "Finn: A framework for fast, scalable binarized neural network inference". In: *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pp. 65–74.

Umuroglu, Yaman et al. (2018). "BISMO: A scalable bit-serial matrix multiplication overlay for reconfigurable computing". In: *2018 28th International Conference on Field Programmable Logic and Applications (FPL)*. IEEE, pp. 307–3077.

Varma, Ayush et al. (2018). "House Price Prediction Using Machine Learning and Neural Networks". In: *2018 Second International Conference on Inventive Communication and Computational Technologies (ICICCT)*. IEEE, pp. 1936–1939.

Vasudevan, Aravind et al. (2017). "Parallel multi channel convolution using general matrix multiplication". In: *2017 IEEE 28th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*. IEEE, pp. 19–24.

Velasco-Montero, Delia et al. (2019). "On the Correlation of CNN Performance and Hardware Metrics for Visual Inference on a Low-Cost CPU-based Platform". In: *2019 International Conference on Systems, Signals and Image Processing (IWSSIP)*. IEEE, pp. 249–254.

Waibel, Alex et al. (1989). "Phoneme recognition using time-delay neural networks". In: *IEEE transactions on acoustics, speech, and signal processing* 37.3, pp. 328–339.

Wang, Dayong et al. (2016a). "Deep learning for identifying metastatic breast cancer". In: *arXiv preprint arXiv:1606.05718*.

Wang, F. et al. (2016b). "Where does AlphaGo go: from church-turing thesis to AlphaGo thesis and beyond". In: *IEEE/CAA Journal of Automatica Sinica* 3.2, pp. 113–120.

Werbos, Paul (1974). "Beyond Regression: New Tools for Prediction and Analysis in the Behavioral Sciences". In: *Ph. D. dissertation, Harvard University*.

Widrow, Bernard (1987). "Adaline and Madaline". In: *Proc. IEEE 1st Int. Conf. on Neural Networks*. Vol. 1, pp. 143–57.

Widrow, Bernard and Marcian E Hoff (1960). *Adaptive switching circuits*. Tech. rep. Stanford Univ Ca Stanford Electronics Labs.

Wikipedia (2020). *Automata Theory*.

Wolf, C. et al. (2013). "Yosys - A Free Verilog Synthesis Suite". In: *Proceedings of the 21st Austrian Workshop on Microelectronics*. Austrochip 2013. Austria: Springer.

Xiong, Hui Y et al. (2015). "The human splicing code reveals new insights into the genetic determinants of disease". In: *Science* 347.6218, p. 1254806.

Xu, Qiang et al. (2015). "Approximate computing: A survey". In: *IEEE Design and Test* 33.1, pp. 8–22.

Xu, S. and D. Gregg (2017). "Bitslice Vectors: A Software Approach to Customizable Data Precision on Processors with SIMD Extensions". In: *2017 46th International Conference on Parallel Processing (ICPP)*. USA: IEEE, pp. 442–451.

Zaidy, Aliasger (2016). "Accuracy and Performance Improvements in Custom CNN Architectures". In: *Purdue University*.

Zeiler, Matthew D and Rob Fergus (2013). "Visualizing and understanding convolutional networks". In: *European conference on computer vision*. Springer, pp. 818–833.

Zeng, Haoyang et al. (2016). "Convolutional neural network architectures for predicting DNA - protein binding". In: *Bioinformatics* 32.12, pp. i121–i127.

Zhang, Chen et al. (2015). *Optimizing FPGA-based Accelerator Design for Deep Convolutional Neural Networks*.

Zhang, Tianhao et al. (2016). "Learning deep control policies for autonomous aerial vehicles with mpc-guided policy search". In: *2016 IEEE international conference on robotics and automation (ICRA)*. IEEE, pp. 528–535.

Zhou, Jian and Olga G Troyanskaya (2015). "Predicting effects of noncoding variants with deep learning-based sequence model". In: *Nature methods* 12.10, p. 931.