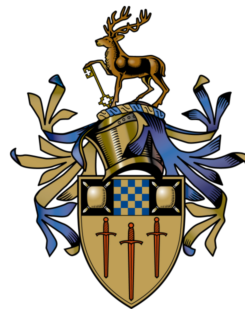
Optimal Use of Machine Learning for Planetary Terrain Navigation

Peter Charles Blacker

Submitted for the Degree of
Doctor of Philosophy
from the
University of Surrey



Surrey Space Centre
Faculty of Engineering and Physical Sciences
University of Surrey
Guildford, Surrey GU2 7XH, U.K.

May 2020

© Peter Charles Blacker 2020

Abstract

Machine Learning (ML) is spreading into more application areas and facilitating a step change in autonomous perception and comprehension capabilities. Within the space sector it is currently used in the ground segment and its use in the space segment is being actively investigated. ML is especially good at perception tasks that have traditionally been difficult for computers to master, improvements in these perception capabilities have facilitated a wide range of new terrestrial applications such as autonomous vehicles and drones as well as language comprehension and translation. Deep space probes and rovers are reliant upon their on-board autonomy since communication opportunities are sporadic, low bandwidth, and high latency. The levels of autonomy these craft have directly affects their capabilities, enabling them to perform activities without direct commands from ground controllers.

ML models have the potential to increase the level of spacecraft autonomy, expanding mission capabilities, science returns, and returns on investment however two formidable barriers to adoption exist. Firstly confidence in ML as a discipline and of the performance of specific models is lower than that usually expected in the aerospace community, careful mission design is required to demonstrate and especially utilise ML on active space missions. Secondly limited processing power of space qualified radiation tolerant processors presents challenges not seen in many terrestrial applications to date. On-board anomaly detection and robotic perception are two applications where the use of ML on-board space vehicles and rovers is currently undergoing active research and development.

To the authors knowledge this thesis presents the first investigation into the use of ML for the estimation of terrain navigability on-board planetary rovers. The suitability of both Convolutional Neural Network (CNN)s and encoder-decoder models is evaluated in terms of their accuracy and computational performance using two planetary terrain data-sets. Their accuracy was found to match that of existing state of the art navigability estimators, while surpassing their performance. Deployment of ML models onto radiation hardened processors is identified as barrier to adoption, since no software tools existed which targeted these processors.

Experimental tools automating the deployment and validation tasks are developed, which are the first of their kind to target radiation hardened processors. Enabling new insights in the study of low level ML implementation on current and next generation radiation hardened processors. Using these tools novel techniques are found that significantly reduce the amount of memory required to perform inference on a wide range of contemporary benchmark models, while using state of the art techniques to optimise execution time. The memory requirement of MobileNet v1 is reduced by 33% while MobileNet v2 is reduced by 53%. The impacts of new techniques are been analytically characterised, and automated tools developed which allow rapid evaluation and adoption in the wider ML community.

New techniques discovered during this work are informing the current development of the Mars Sample Fetch and Return rover at Airbus as well as other machine learning groups in the space industry. These tools are enabling transfer of existing techniques from the ML community into the space sector. While these methods have been developed for space applications on radiation hardened processors, they are equally applicable to low power terrestrial computing. The spread of ML onto micro-controllers in embedded Internet of Things (IoT) devices is using these techniques impacting the performance a wide range of applications outside of the space sector.

Key words: Planetary Rover, Cost-mapping, Machine Learning, Optimisation, Autonomy

Email: Pete.Blacker@Gmail.com

WWW: https://www.linkedin.com/in/pete-blacker

Acknowledgements

I would like to thank my supervisors at Surrey University, Chris Bridges and Simon Hadfield, for stepping up and guiding me through this PhD when I needed support. The work presented would not have been possible without their help. My thanks also goes to my supervisors at Airbus who have kept my work grounded in reality, and provided unique insights into the Exomars rover, Mattias Winter, Anton Donchev, and Piotr Weclewski. All the staff and students at Surrey Space Centre, I have enjoyed my time here immensely, and have been introduced to the fascinating world of putting things in space and hoping they don't break!

Most importantly of all I express my deep gratitude to my partner Donna who has supported me so much during these four years. Even when you had to concentrate on the, slightly more important, task of looking after our son. I can not thank you enough for motivating me at my low points, keeping me sane when it all seemed impossible, and being there for me regardless how tired, grumpy or incoherent I was!

Declaration

This thesis and the work to which it refers are the results of my own efforts. Any ideas, data, images or text resulting from the work of others (whether published or unpublished) are fully identified as such within the work and attributed to their originator in the text, bibliography or in footnotes. This thesis has not been submitted in whole or in part for any other academic degree or professional qualification. I agree that the University has the right to submit my work to the plagiarism detection service TurnitinUK for originality checks. Whether or not drafts have been so-assessed, the University reserves the right to require an electronic version of the final document (as submitted) for assessment as above.

The work presented in this thesis is also present in the following manuscripts:

- P Blacker, CP Bridges, and S Hadfield. Rapid prototyping of deep learning models on radiation hardened cpus. In *2019 NASA/ESA Conference on Adaptive Hardware and Systems (AHS)*, pages 25–32. IEEE, 2019.
- Blacker, P., Bridges, C.P. and Hadfield, S., 2020. Diagonal Memory Optimisation for Machine Learning on Micro-controllers. arXiv preprint arXiv:2010.01668.

Signed:

Date: 25th October 2020

Contents

Nomenclature **xi**

List of Figures **xv**

List of Tables **xxi**

1 Introduction **1**

1.1 Research Motivation 3

1.2 Research Scope 3

1.3 PhD Aims and Objectives 4

1.4 Research Contributions 4

1.5 Publications and Releases 5

1.6 Overview of Thesis 5

2 Literature Review **7**

2.1 Introduction 7

2.2 GNC Architectures 8

2.2.1 Limitations of Telecommand 8

2.2.2 Jet Propulsion Laboratory (JPL) Rover Guidance Navigation and Control (GNC) Architectures 9

2.2.3 Archiecture of the Exomars Rosalind Franklin Rover 11

2.3 Rover Nagivation Sensors 14

2.3.1 Steroposis 15

2.3.2 Lidar Sensing 18

2.4 Navigation Cost-map Generation 20

2.4.1 Cost-map Generation Techniques 22

2.5 Machine Learning Onboard Spacecraft 24
2.5.1 Software Development and Validation & Verification 26
2.5.2 Computational Power Limitations 27
2.5.3 Deployment Tools 32
2.6 Gaps in Knowledge 35

3 Industrial Problem & ML Solution **37**
3.1 Introduction 37
3.2 Industrial Problem Definition 39
3.2.1 Existing Techniques 40
3.3 ML Model Analysis 41
3.3.1 ML Models Evaluated 43
3.3.2 Model Evalulation 46
3.3.3 Datasets 47
3.3.4 Pre-processing and Data Augmentation 51
3.3.5 Absolute Elevation Invariance 54
3.3.6 Model Training 56
3.3.7 Initial Inference Timing 59
3.4 Inference Feasablilty on LEON Processors 60
3.4.1 LEON Deployment Using Existing Tools 61
3.4.2 Tools Released During this Work 63
3.5 Results 64
3.5.1 Effects of Model Scale 66
3.5.2 Costmapping Performance Results 67
3.5.3 Comparison of Model Accuracies on Different Terrains 68
3.6 Summary 70
3.6.1 Future Work 71

4 TFMin Tool **73**
4.1 Introduction 73
4.2 Motivation 74
4.3 Architecture 74

4.3.1 Design Rquirements 76
4.3.2 Graph Representation 77
4.3.3 Graph Translation Pipeline 78
4.3.4 Using Graph-Translators for Introspection 80
4.3.5 Tensor Memory Model 80
4.3.6 Operation Kernels 84
4.3.7 Memory Optimisation 86
4.4 Analysing Deployed Models 89
4.4.1 Memory Requirements Analysis 89
4.4.2 Detailed Memory Access Analysis 90
4.4.3 Analysing the Output of Generated Implementations 93
4.5 Use Cases 96
4.5.1 Layer Implementation Performance Analysis 97
4.5.2 Memory Optimisation 97
4.5.3 Computationl Requirements of Cost-Mapping Models 98
4.6 Summary 103

5 Memory Optimisation 107
5.1 Introduction 107
5.2 Problem Definition 109
5.2.1 Effects on Power and Latency 111
5.3 Existing Approach to Reducing Peak Memory Use 112
5.3.1 Tensor Buffer Reuse 112
5.4 Novel Techniques for Memory Optimisation 115
5.4.1 Operation Splitting 115
5.4.2 Diagonal Memory Optimisation 121
5.4.3 Calculating the Safe Buffer Overlap 123
5.5 Results 137
5.5.1 Sequential Published Models 137
5.5.2 Connected Published Models 140
5.5.3 Cost-Mapping Models 142
5.6 Summary 144

6 Conclusion and Future Work **149**

6.1 Future Work . . . 152

A Detailed Cost-Mapping Model Descriptions **155**

A.1 Cnn-A Model . . . 155

A.2 Cnn-B Model . . . 157

A.3 Cnn-C Model . . . 159

A.4 Cnn-D Model . . . 161

A.5 Cnn-E Model . . . 163

A.6 EncDec-A Model . . . 165

A.7 EncDec-B Model . . . 167

A.8 EncDec-C Model . . . 169

A.9 EncDec-D Model . . . 171

A.10 EncDec-E Model . . . 173

A.11 EncDec-F Model . . . 175

A.12 EncDec-X Model . . . 177

B Memory Optimisation Appendix **179**

B.1 Tensorflow Lite Reference Operations . . . 179

Bibliography **183**

Nomenclature

AHS Adaptive Hardware and Systems

ML Machine Learning

DL Deep Learning

TF Tensorflow

TFL Tensorflow Lite

TFLμ Tensorflow Lite Micro

NN Neural Network

CNN Convolutional Neural Network

RNN Recurrent Neural Network

XLA Accelerated Linear Algebra

AOT Ahead of Time

RL Reinforcement Learning

ONNX Open Neural Network Exchange

AMC AutoML for Model Compression

MSE Mean Squared Error

SVM Support Vector Machine

TPU Tensor Processing Unit

DMO Diagonal Memory Optimisation

VMT Visual Memory Tracer

DEM Digital Elevation Model

FOV Field of View

SLAM Simultaneous Localisation and Mapping

LIDAR Light Detection and Ranging

GESTALT Grid-based Estimation of Surface Traversability Applied to Local Terrain

GPR Ground Penetrating Radar

SPA Sense Plan Act

VO Visual Odometry

WAC Wide Angle Camera

IMU Inertial Measurement Unit

ESA European Space Agency

NASA National Air and Space Administration

JPL Jet Propulsion Laboratory

MER Mars Exploration Rovers

MSL Mars Science Laboratory

V&V Verification and Validation

LEO Low Earth Orbit

LMO Low Mars Orbit

OBDH On Board Data Handling

GNC Guidance Navigation and Control

TRL Technology Readiness Level

ERGO European Robotic Goal-Oriented Autonomous Controller

SSC Surrey Space Centre

DSN Deep Space Network

SFR Sample Fetch Rover

MAV Mars Ascent Vehicle

AIT Assembly Integration and Test

RTG Radioisotope Thermoelectric Generator

TID Total Ionising Dose

BAE British Aerospace Marconi Electronic

ECSS European Cooperation for Space Standardization

EO Earth Observation

SAA South Atlantic Anomaly

COTS Commercial of the Shelf

GCR Galactic Cosmic Rays

PROBA Project for On-Board Autonomy

MDA Macdonald, Dettwiler And Associates

OBDH On-board Data Handling

ISRO Indian Space Research Organisation

OBC On-board Computer

CI Confidence Interval

CB Confidence Band

ARM Advanced RISC Machines

IEEE Institute of Electrical and Electronics Engineers

ICP Iterative Closest Point

CMOS Complimentary Metal Oxide Silicon

SEU Single Event Upset

SET Single Event Transient

SEL Single Event Latchup

SEB Single Event Burnout

TDC Time to Digital Converter

RAM Random Access Memory

SRAM Static Random Access Memory

SDRAM Synchronous Dynamic Random Access Memory

EEPROM Electronically Erasable Programmable Read Only Memory

DSP Digital Signal Processor

CPU Central Processing Unit

GPU Graphics Processing Unit

FPU Floating Point Unit

FLOP Floating Point Operation

FLOPS Floating Point Operations per Second

MIPS Million of Instructions per Second

FIFO First In First Out

SIMD Single Instruction Multiple Data

FPGA Field Programmable Gate Array

LLVM Low Level Virtual Machine

IR Intermediate Representation

DMA Direct Memory Access

MAC Multiply-Accumulator

XML Extensible Mark-up Language

RGB Red Green Blue

ANSI American National Standards Institute

API Application Programming Interface

GCC GNU Compiler Collection

IoT Internet of Things

GPL GNU General Public License

CUDA Compute Unified Device Architecture

OS Operation System

FDIV Floating Point Divide

JIT Just in Time

List of Figures

2.1 Points inside the maximal rover footprint which are isolated and used to calculate geometric terrain metrics. Rover image courtesy of edupics.com. 11

2.2 Goodness (Cost) map generated by Spirit on sol 107 of its mission. Red areas are impassible and yellow/green are traversable with differing levels of ease. Taken from Biesiadecki et al. [21] 12

2.3 Architecture of the Exomars rover on-board GNC systems, courtesy of Airbus [154]. 13

2.4 Stereo depth estimation errors of the Mars Science Laboratory (MSL) naviation and hazard cameras, taken from Maki et al [99]. 17

2.5 Comparison of navigation camera depth errors for MSL Curiosity and Exomars, full and half resolution disparity maps, based on published values from [99] [129]. 18

2.6 Opto-mechanical design of the fast scanning Light Detection and Ranging (LIDAR) developed by Bakambu et al. taken from [13]. 19

2.7 Example navigation map produced using obstacle expansion. Obstacles (blue) have been expanded by the radius of the maximal rover footprint into hazard areas (red), the resulting map can be used to easily plan safe routes by finding the shortest line which does not enter the hazard area (green). 21

2.8 Exomars on-board mapping perception pipeline which captures pan cam stereo pairs and processed them into navigation route plans towards the requested goal. Courtesy of Airbus [154]. 23

2.9 RAD750 single board computer currently in use by the MSL curiosity rover on Mars, courtesy of British Aerospace Marconi Electronic (BAE) systems. 31

2.10 The GR712RC radiation hardened space processor, courtesy of Gaisler Aeroflex. 31

2.11 Block diagram of the GR740 quad core LEON 4 radiation hardened microprocessor. Courtesy of Gaisler Aeroflex. 32

2.12 Labelled chip plot of the radiation hardened RAD5545 quad core space processor, courtesy of BAE Systems. 33

3.1 (top) Digital elevation map taken from the Airbus Marsyard. (bottom) Generated navigation cost-map of values, white border areas are of unknown cost. 40

3.2 High level processing steps of our ML terrain estimator evaluation experiment. 42

3.3 High level description of the five CNN models evaluated in our Adaptive Hardware and Systems (AHS) publication [22]. Detailed descriptions of these models can be found in Appendix A 44

3.4 Topology of an encoder decoder model E used in this work. Here two transposed convolution layers with 5×5 filters and strides of 2 have been used to expand the output of the model. 45

3.5 a) 2D histogram of ML estimates against true cost-values. b) Confidence bands of navigability estimates. 47

3.6 The Mars Yard test terrain at Airbus Stevenage. 48

3.7 Low resolution Digital Elevation Model (DEM) of the Mars Yard generated using photogrammetry. Note the lack of fine detail, especially around the edges of the map. 49

3.8 Custom made Lidar camera fusion sensor, used for detailed mapping of the Stevenage Mars Yard. 50

3.9 High resolution DEM of the Mars Yard generated from LIDAR data. Rocks and edges are much more clearly defined than the earlier map generated using photogrammetry. 50

3.10 Terrain DEM from the European Robotic Goal-Oriented Autonomous Controller (ERGO) field trial site, covering an area of approximately 300 x 300 metres. Two sub regions of the ERGO terrain map have been used for model training and evaluation. 'Slope Hill' region is shown in blue, 'River Bed' region is shown in green. 51

3.11 Ergo 'River Bed' terrain on the left, 'Slope Hill' terrain on the right. 52

3.12 Extraction of a single training pair from a registered DEM and navigation cost map. Where O_e is the output edge length. 52

3.13 Rotation augmentation of the Mars Yard lidar dataset, with rotations from 9 to 81 degrees. Additional augmentation using reflections and cardinal rotations is performed online during training. 53

3.14 Navigation estimate confidence bands for the encoder-decoder-A model trained on all four test terrains, with either no elevation invariance, weight normalisation, or layer normalisation. It can clearly be seen that layer normalisation produces consistently better models, although weight normalisation is also an improvement over the baseline. The models trained on maps from the ERGO dataset failed to converge at all without one of elevation invariance techniques being applied. 55

3.15 Comparison of elevation invariance pre-processing techniques on all four training maps. A, shows the mean error at 3 sigma. B, shows the worst error at 3 sigma. 56

3.16 Map of error regions divided by their effects. 59

3.17 2D Histogram and confidence bands of estimates against training values for the Encoder Decoder topology D trained for 5000 steps. 64

3.18 Cost map estimated using a trained Encoder Decoder model on the left in comparison with the ground truth cost map on the right. 65

3.19 3 sigma error results for all CNN models and scales. Encoder Decoder results are shown in grey for comparison. 66

3.20 3 sigma error results for all Encoder Decoder models and scales. CNN results are shown in grey for comparison 66

3.21 GPU performance measurements of all models collected on an Nvidia GTX-1070. 68

3.22 3 sigma error averaged across all model scales, shown for each training dataset and three model topologies. 69

4.1 Data flow of ML model deployment using TFMin, showing the three top level processes performed. 75

4.2 Data structure of the three Core objects TFMin uses to represent ML models internally. In the interests of clarity only high level attributes are shown. 78

4.3 Example of (a) memory and (b) graph visualisations generated generated after operation-splitting optimisation, see Section 5.4.1. Operation and Tensor highlights have been added by the algorithm to clearly identify the parts of the model's graph which were split into parallel chains. 81

4.4 SqueezeNet intermediate buffer layout within a 6.3 MB tensor arena. Buffer locations computed using a forwards pass of a heap algorithm. 82

4.5 Example of tensor-buffer reuse which is made possible by non-contiguous memory addressing. 83

4.6 SqueezeNet intermediate buffer layout, showing how the layout generated using a heap algotithm shown in Figure4.4, can be further optimised to use less memory. 88

4.7 Pre-allocation pattern for a deployment of the Inception Resnet V2 model. Buffers which define the tensor arena size are highlighted in blue. 90

4.8 Information flow diagram of a binary being traced by the Visual Memory Tracer. 92

4.9 Full memory trace of a Diagonal Memory Optimisation (DMO) optimised deployment of Mobile Net using TFMin. High resolution images are produced by Visual Memory Tracer (VMT) to better capture the fine detail of the memory trace. These do not work well in print, so zoomed areas are provided to reveal the details of the tensor operations. 94

4.10 The five terrain assessment models analysed in [22], convolution layers shown in yellow and fully-connected layers shown in blue. Filter/weight sizes indicated under each layer. 97

4.11 Inference execution time results of the EncDec-A model. Desktop Graphics Processing Unit (GPU) results shown in milliseconds on the left and single core 200 MHz LEON3 results shown in seconds on the right. 100

4.12 Multiply-Accumulator (MAC) operation count of the EncDec-A model compared to the execution time of this model on a LEON3. 100

4.13 Inference execution time results of all cost-mapping models. Desktop GPU results shown on the left and single core 200 MHz LEON3 results shown on the right. 101

4.14 Accuracies of all models with the Encoder-Decoder model X highlighted shown in comparison to the performance of all models on a single core 200 MHz LEON3. 102

5.1 Intermediate tensor buffer locations for MobileNet v1 0.25 128, 8 bit quantised. Location within the tensor arena is shown on the x-axis while the scope of each buffer from first to last use is shown on the y-axis. 109

5.2 a, Sub-graph showing the 3rd fire module of the Squeezenet model. b, Intermediate buffer allocations of this model. It can be seen that the buffers of 3rd fire module (highlighted in blue) define the 6428 KB peak memory requirement of this model. 113

5.3 a, 3rd fire module of Squeezenet where the 2D convolution operations write directly into the super-tensor. b, Intermediate buffer allocations of this model. Buffers of the 3rd fire module are highlighted in blue. It can be seen that the optimised fire module no longer defines peak memory requirement of the model, it is now set by the the first max-pooling operation at 5838 KB. 114

5.4 a, Subset of MobileNet showing the 2nd and 3rd operations before optimisation. b, Equivalent subset of the optimised graph, which computes the output tensor using five parallel pairs of operations. Note that there is a necessary overlap between the three intermediate tensors and input tensor slices because of the overlapping receptive fields of the depth-wise convolution operation. 116

5.5 a, Intermediate buffer allocations of a full sized MobileNet V2 implementation. b, Intermediate buffer allocations of the same model in which the second and third operations have been split into five parallel branches. Buffers that have been effected by this optimisation are highlighted in blue. 117

5.6 Intermediate buffer memory access pattern for the example model (MobileNet v1 0.25 128 quantised). In use areas shown in grey, load, store, and update events in red, blue, and green respectively. Plot a shows the memory access pattern when the original heap allocation strategy is used to allocate intermediate buffers, large areas of unused memory can be see which could be used to reduce the size of the tensor arena. Plot b shows the memory access pattern of the same model with intermediate buffers allocated using diagonal memory optimisation, in-use memory is packed more densely allowing the size of the tensor area to be reduced. 122

5.7 Memory traces of four common ML tensor operations. (a) Rectified Linear Unit, (b) Matrix Multiplication, (c) Depthwise Convolution, (d) 2D Convolution. These traces only show intermediate input & output tensor buffers, ignoring the filter and weight buffers. 124

5.8 Definition of the safe buffer overlap (O_s) metric, defined as the maximum overlap where no in-use areas of memory are clobbered. In use memory shown in grey, write operations in red, and read operations in blue. 125

5.9 Memory read pattern example from a depthwise 2D convolution. Points highlighted define a linear boundary containing all read operations. 131

5.10 $minR(i)$ bounding function for the depthwise 2D convolution implementation. It can be seen that all read operations (in blue) lie above the monotonic function (green). 133

5.11 The two possible definitions of the analytical minimum bound, depending on the relative gradient of the $minR$ & $maxW$ functions. 133

5.12 Memory trace of a 5×5 2D Convolution operation being executed using four threads. 136

5.13 a, Original buffer allocation pattern of MobileNet v1 2.0 224. b, Optimised buffer allocation pattern after operation-splitting has been applied to the graph. 138

5.14 a, Original buffer allocation pattern of MobileNet v2 0.35 224. b, Optimised buffer allocation pattern after operation-splitting has been applied to the graph. 139

5.15 a, Inception v4 original buffer allocation pattern (only the first third of the model is shown for clarity). b, Buffer allocation pattern produced using operation-splitting, split tensors shown in yellow, final output of split block shown in green. Note that in this operation-splitting optimisation the input buffer is the start of the split, so is not shown in this figure. 141

5.16 a, DenseNet original buffer allocation pattern (only the first fifth of the model is shown for clarity). b, Buffer allocation pattern optimised using DMO, Peak memory defining buffers are shown in blue. It can be seen that none of the peak memory defining buffers of the optimised pattern use DMO to overlap. 142

5.17 Original buffer pre-allocation patterns of: a, Model EncDec-F. b, Model EncDec-X. Intermediate buffers defining the peak memory requirement are highlighted in blue. 142

5.18 Initial section of the tensor graph of Encoder Decoder model X after modification by the operation-splitting memory optimiser. Showing the first two operations split into five parallel chains. 143

List of Tables

2.1 Opportunities identified during this literature review. 36

3.1 Set of eleven model topologies trained and analysed during our work. 45
3.2 Scalar metrics for example model 47
3.3 Sizes of training datasets with cardinal rotation augmentation and arbitrary rotation augmentation. 53
3.4 Opportunities at the end of the cost mapping investigation. 72

4.1 Memory requirements of proposed cost-mapping models. 103
4.2 Opportunities at the end of the cost mapping investigation. 105

5.1 Possible operation splitting optimisations found using algorithm. 121
5.2 Specification of 2nd Depthwise 2D Convolution in MobileNet 135
5.3 Estimation Error of Safe Overlap (O_s) 135
5.4 Reduction in memory requirements of sequential published models 139
5.5 Reduction in memory requirements of connected published models 141
5.6 Reduction in required memory for inference of proposed cost-mapping models 144
5.7 Opportunities at the end of the cost mapping investigation. 147

6.1 Opportunities and proposed solutions presented in this thesis. 151

A.1 Detailed description of Cnn-A model. 156
A.2 Detailed description of Cnn-B model. 158
A.3 Detailed description of Cnn-C model. 160
A.4 Detailed description of Cnn-D model. 162
A.5 Detailed description of Cnn-E model. 164
A.6 Detailed description of EncDec-A model. 166

A.7 Detailed description of EncDec-B model. 168
A.8 Detailed description of EncDec-C model. 170
A.9 Detailed description of EncDec-D model. 172
A.10 Detailed description of EncDec-E model. 174
A.11 Detailed description of EncDec-F model. 176
A.12 Detailed description of EncDec-X model. 178

Chapter 1

Introduction

To date four robotic rovers have successfully landed and explored the surface of Mars, increasing our knowledge and understanding of the red planet considerably while throwing up many fascinating new questions. Unexpected sub-annual variances in atmospheric methane detected by the MSL Curiosity, are a tantalising indication of past or present microbial life [149] [74]. These discoveries are made possible by the defining capability of these rovers to travel across Martian terrain in search of varied locations for scientific study.

It follows that the greater distance these rovers are capable of travelling, the more scientific targets can be visited increasing overall knowledge gained and value of a mission. However due to the manifold challenges of operating at a distance of three to twenty two light minutes on unknown terrain, the traverse speeds of these rovers are incredibly low. The JPLs solar powered Mars Exploration Rovers (MER) Spirit and Opportunity had a maximum speed of 3.75 cm per second [21]. While the Radioisotope Thermoelectric Generator (RTG) powered Curiosity rover has a maximum speed of 4.2 cm per second [119].

The autonomous GNC systems of these rovers all use the sense-plan-act architecture [128] due to the limited processing capabilities of the radiation hardened processors used [2] [133]. While operating in autonomous drive mode these rovers spend a significant time stationary, while complex perception algorithms execute on board. The distance they are capable of travelling in a sol is limited not only by the speed of their locomotion system but also the speed of their on-board computers and GNC algorithms.

These limitations of autonomy are the reason that direct teleoperation is still commonplace for these rovers, MSL has been mostly operated in ’blind drive‘ mode [119], and both the MER rovers used a combination of blind drive and autonomous mode on each sol [21]. Direct operation of rovers in this way increases demands on ground controllers and the Deep Space Network (DSN) used to communicate with them. This network already runs at capacity, so adding to its workload results in less bandwidth for other deep space probes.

In the mean time back on Earth where processing power is far more abundant, the field of Machine Learning has increased the capabilities of many autonomous algorithms from machine translation [41], image classification [35] to obstacle detection [120]. Two of the most computationally demanding autonomy tasks performed on-board Mars rovers are stereo disparity map generation and navigation cost-map generation. ML has been proposed to improve both of these tasks for terrestrial autonomous vehicles [104] [150]. The goal of this work is to use ML to increase the autonomy level of the current generation of Mars rover GNC systems, allowing greater distances to be traversed with a lower demand on the DSN and ground controllers.

The National Air and Space Administration (NASA)s Mars 2020 rover Perseverance is currently in its cruise phase on the way to the red planet carrying sample cannisters which will hopefully be returned to earth by future missions. In Europe the Exomars rover ’Rosalind Franklin‘ is undergoing final Assembly Integration and Test (AIT) in preparation for launch during the 2022/2023 window. European Space Agency (ESA) is currently contracting early design and development work for the Sample Fetch Rover (SFR), as part of the Mars sample return mission. Unlike rovers which have gone before it, the SFRs primary goal is not scientific but logistic, it’s task is to collect sample canisters deposited by Perseverance and transport them to the Mars Ascent Vehicle (MAV) for launch into Low Mars Orbit (LMO). As such the speed at which it can traverse Martian terrain is a more important design driver than it has been for previous rovers.

The SFR is an ideal candidate for novel autonomous GNC algorithms which shorten the stationary phases of Sense Plan Act (SPA) autonomy, thereby increasing the distance travelled per sol. Increasing the autonomy of this rover will also reduce demand on the DSN which may be required to shares its resources between an unprecedented four rovers at the time, depending on their lifetimes. The inclusion of ML modules within a GNC system could increase the speed

rovers can cover Martian terrain, and prove a technology for use on future robotic explorers.

This research presents an ML solution to the cost-map generation task performed by planetary rover GNC systems. Analyses are performed studying the accuracy and computational cost of this approach on desktop and representative flight processors. A new tool has been developed that allows the automated conversion of high level ML models into American National Standards Institute (ANSI) C code suitable for use in the flight software development process. Novel optimisation algorithms have been presented and implemented using this tool which reduce the amount of Random Access Memory (RAM) required to perform inference. These optimisations increase the feasibility of implementing the cost-mapping models presented on the space qualified Central Processing Unit (CPU)s required for the SFR mission.

1.1 Research Motivation

There are two factors which limit the speed Mars rovers are able to cross terrain, and neither of these is the speed of their locomotion systems. Under direct tele-operation the light delay, and bandwidth of the DSN limits opportunities for manual routes to be sent. While in autonomous driving modes the speed of perception algorithms limits the amount of time the rover is in motion.

Nothing can be done about the round trip light delay, and short of investing in more dishes the capacity of the DSN will remain fixed. However we see an opportunity to increase the autonomy level of Mars rovers, thereby increasing the capabilities of these missions while reducing dependence upon the resources of the DSN and ground controllers.

1.2 Research Scope

This research is focussed on bringing recent advances in the field of ML to the application of autonomous GNC systems on-board planetary rovers. There are many hurdles to overcome before this technology is flown, our work contributes an increase in the Technology Readiness Level (TRL) of these techniques and argues they are compelling enough to warrant further investigation. Our specific application is the Mars Sample Fetch rover. Although our findings

are equally applicable to any surface or orbital mission in deep space, which will benefit from the deployment of ML.

1.3 PhD Aims and Objectives

The initial aim of this work is to improve the current autonomy capabilities of Mars rovers, focussing specifically on the challenges of the SFR currently being developed by ESA and NASA. This objective is broken down as follows:

- Identify opportunities for ML to improve the on-board cost-map generation task and demonstrate the suitability of this new approach.
- Investigate the challenges of deploying ML models to the LEON family of radiation hardened processors which will be used on the SFR rover.
- Discuss the challenges which remain before this novel technique could be used on an actual space mission and propose tools to aid adoption.

1.4 Research Contributions

This research contributes an analysis of ML cost-mapping models, presenting tools and novel optimisation algorithms for the deployment of any ML model to small CPU targets. These contributions can be summarised as:

1. A range of ML models for generating navigation cost-maps have been presented. These models are shown to produce maps as accurate as state of the art algorithms while exceeding their performance. Factors affecting the accuracy and computational cost of this approach have been analysed.
2. A new tool has been developed and released open source to the community which can convert ML models into platform independent ANSI C code. This code can be used for research into the deployment process, or as an engineering tool in an industrial setting. This is currently the only ML deployment tool which generates ANSI C code and creates

the most lightweight implementations of the new generation of edge ML tools. This tool is enabling research and development of ML solutions within the space sector which was not possible with the tools which preceded it.

3. After discovering that larger navigation cost models are more computationally efficient, two novel memory optimisation algorithms are presented that significantly reduce the amount of RAM required to perform inference. These algorithms outperform the current state of the art deployment tools, whilst being complementary to existing model compression techniques.

1.5 Publications and Releases

The work presented in this thesis has been published in the following papers, and released open-source to the community.

- Blacker, P., Bridges, C.P. and Hadfield, S., 2019, July. Rapid Prototyping of Deep Learning Models on Radiation Hardened CPUs. In 2019 NASA/ESA Conference on Adaptive Hardware and Systems (AHS) (pp. 25-32). IEEE.
- Blacker, P., Bridges, C.P. and Hadfield, S., 2020. Diagonal Memory Optimisation for Machine Learning on Micro-controllers. arXiv preprint arXiv:2010.01668.
- TFMin Tensorflow deployment tool is available under the GNU General Public License (GPL) v3 licence from [https://github.com/PeteBlackerThe3rd/TFMin]

1.6 Overview of Thesis

Chapter 2 reviews the current state-of-the-art GNC systems used on-board Mars rovers from NASA and ESA. The level of autonomy available to ground controllers is discussed along with its used in practice. Sensors and algorithms used by these rovers to perceive the terrain around them are discussed, along with downstream cost-mapping algorithms which facilitate autonomous route planning. The new field of terrestrial ML cost-map generation is introduced and

its suitability for use on planetary robotics discussed. Finally the use of ML in the space sector is described, including examples of its use on-board spacecraft and technical and operational challenges that still remain.

Chapter 3 describes our analysis of using ML models to produce navigation cost-maps on-board planetary rovers. We initially study the problem itself and existing solutions used on previous Mars rovers, then define our experimental set up including training datasets and evaluation strategy. The performance of a set of twelve different ML topologies are given in terms of model size and the accuracy of navigation cost estimates. This demonstrates the feasibility of this approach and identifies a number of opportunities for future improvements.

Chapter 4 addresses the challenge of deploying ML onto the low power radiation hardened CPUs required for deep space missions. This is achieved using a new code generation tool which converts high level ML models described in Tensorflow to platform independent ANSI C code. An open and extensible tool is presented which allows the process to be optimised and adapted to specific applications. Importantly the code generation approach allows this method to fit into the existing software Verification and Validation (V&V) practices used in the space industry.

Chapter 5 presents two novel optimisation algorithms which can significantly reduce the amount of RAM required when performing inference using ML models. These two algorithms are demonstrated on a range of state of the art models and the terrain evaluation models proposed in Chapter 2. These new algorithms are not just applicable to the radiation hardened processors used in space but also to the emerging area of 'Edge ML' whereby embedded micro-controllers on the periphery of networked systems are starting to implement ML inference.

Chapter 6 concludes this thesis summing up our findings and impact. Recommendations are made for promising future areas of research and engineering which can raise the TRL level of this technology and get ML closer to performing inference on the surface of Mars.

Chapter 2

State-of-the-Art

2.1 Introduction

This work aims to improve the speed with which planetary rovers are able to understand their environment and plan safe routes across it. This task currently consumes the majority of CPU time for the GNC process and is largely responsible for the slow speeds at which rovers move across the Martian surface. The technology used on the current generation of Mars rovers will be reviewed and opportunities to improve the cost-map generation task within the GNC process identified.

A review will be presented showing the latest developments in ML techniques with reference to their potential use in space. Specifically the challenge of deploying these networks on radiation hardened processors and the challenge of implementing them within the context of a space mission. The limited history of ML on-board spacecraft will be reviewed along with the approaches used to fly this technology in the past.

The final summary describes current technological opportunities and identifies gaps in research literature with a focus on increasing the speed of the cost-mapping process. Our goal is to use this improvement in processing efficiency to increase the traverse speed of the next generation of planetary rovers, subsequently increasing the science return and scope of future missions.

2.2 GNC Architectures

The GNC system of a planetary rover is responsible for perceiving its environment, estimating its state within it and controlling actuators to safely traverse the surface and arrive at a defined goal. Current state of the art planetary rover GNC systems are heavily dependent upon their ground segments and human involvement. This dependency is due to the high reliability requirements placed on these missions and the relatively low reliability of the rovers autonomous capabilities when operating in isolation. Their speeds are measured in centimetres per second at best, limiting the number of locations they can visit and therefore the amount of information returned to scientists on Earth. The reason for this literal snails pace is the speed at which the rovers GNC system can perceive the environment and their place within it, limited for the most part by the low computing power of current radiation hardened space processors. Improvements in these autonomous capabilities and their ability to ensure the safety of the rover is a significant area of research with the potential to increase the ground covered and therefore science returned by these missions.

The following sections will provide a high level overview of the GNC systems of JPLs MER and MSL rovers and ESA Exomars rover, Rosalind Franklin. As form follows function the overall architecture of these three rover designs is largely similar, the differences will be described particularly focussing on the mapping and route planning elements which are the focus of this research.

2.2.1 Limitations of Telecommand

The first robotic planetary explorers were the Soviet Lunokhod (moon walker) 1 & 2 rovers launched in 1970 and 1973 respectively [81]. These early rovers were entirely remotely operated by ground controllers on Earth, traversing significant distances across the lunar plains of 9.3 km and 39.1 km. Lunokhod 2's distance has only ever been surpassed by JPLs Opportunity rover over four decades later [26]. The remote operation of these rovers was only possible because of the short round-trip light delay between the Earth and Moon of 2.5 seconds, and the absence of communication black-outs.

Communicating with robotic explorers in orbit around or on the surface of Mars presents far

greater challenges. Round trip delay times between Earth and Mars vary between approximately 9 minutes and 44 minutes. Additionally once per synodic period a solar conjunction occurs preventing all communications with Martian probes. This communications black-out lasts around two weeks, with the precise duration dependant upon orbital paths and solar activity [158].

At the time of writing there are six operational probes in orbit around Mars: Three NASA probes Maven, Mars Reconnaissance Orbiter, and Mars Odyssey. Two ESA probes the Trace Gas Orbiter and Mars Express. One Indian Space Research Organisation (ISRO) probe the Mars Orbiter Mission. The five ESA and NASA probes are able to relay data back to Earth for each other as well as the MSL Curiosity rover on the surface. While this relay capability increases communication opportunities even when eclipsed by Mars, the sheer number of probes which have to share time on the DSN reduces the bandwidth available for each probe.

In summary unlike lunar missions, direction tele-operation of Martian rovers is highly impractical, due to bandwidth limitations, communications latency and blackouts. This has motivated the development of semi-autonomous GNC systems going back to the first Martian rover Sojourner delivered along with NASAs Pathfinder lander [102]. Today Martian rovers are operated in either fully or semi autonomous modes with commands being given in the form of goal locations or terrestrially generated route plans.

2.2.2 JPL Rover GNC Architectures

JPLs most recent three Mars rovers, Spirit, Opportunity, and MSL Curiosity all share the same fundamental locomotion and GNC system design. Common six wheeled rocker-bogie suspension is used [62], with almost identical cameras and camera placement [99]. There are however two significant design differences which affect their autonomous capabilities. The increase in processing power on-board Curiosity provided by switching to the RAD750 [18] from the less powerful RAD6000 [59], and the switch from solar panels as a power source to an RTG.

The use of a more powerful computer coupled with the experience gained from operating the two MER rovers for a decade on Mars has resulted in MSL having more numerous and powerful autonomy modes than its predecessors. MER rovers had three driving modes available [21]:

Autonav, is a fully autonomous mode where the rover assesses terrain maps, then plans and follows routes towards a goal location without the involvement of ground control. Visodom, is a reactive autonomous mode where a route planned on the ground is executed while the rover uses Visual Odometry (VO) to ensure the path is followed correctly and wheel slip and sinkage are kept within safe limits. Finally Blind as the name implies follows pre-planned routes without any checks and is used to perform traverses in known areas and around scientific points of interest.

The MSL rover has a wider range of modes, where the algorithms used in different parts of the GNC system can be controlled individually resulting in 24 possible driving configurations [119]. The VO system can set to Off, VO Full, Slip Check, or VO Auto mode. In VO Auto mode the type of local terrain is assessed and used to switch between VO Full and Slip Check to conserved CPU time. The Four path selection modes are: Directed which is analogous to Blind mode on the MER rovers. Guarded is similar to Directed mode, except hazards are detected and if the rover gets to close to one, the drive is terminated for ground control to intervene. Avoid Keepout-Zones, autonomously drives towards a goal but only avoids manually specified hazards, no on-board cost-map is produced. Finally Autonav is the same on the MER rovers, where the rover traverses terrain fully autonomously.

Both of JPLs rover designs use a version of the Grid-based Estimation of Surface Traversability Applied to Local Terrain (GESTALT) software to process stereo images into navigation cost maps. It should be noted that this is a different process to VO using stereo which computes motion via matching of sparse visual features. Unlike more recent rovers such as the Rosalind Franklin rover (Exomars), the details of this GNC system have been published in detail [21]. A geometric approach is used to generate cost maps from input DEMs, based on three metrics.

First a patch of DEM cells are extracted from a 2.6 meter diameter circular region which encompasses the maximum wheelbase of the rover in all possible orientations Figure 2.1. Three geometric scalar metrics are then computed using this DEM patch, 'Slope', 'Roughness', and 'Step'. Each of these metrics are then scaled using the known capabilities of the rover platform and the costliest value chosen. This algorithm is a form of obstacle expansion [45], so that routes can be found on the resulting cost maps using low cost route planners such as A-Star [64]. GESTALT uses goodness metric which is the inverse of the navigation cost metric used in our

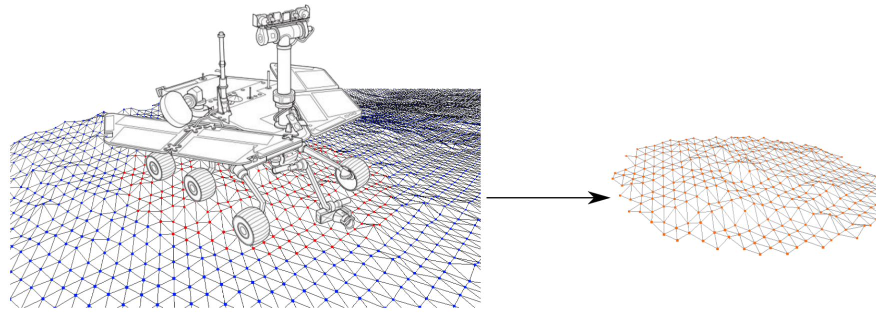


Figure 2.1: Points inside the maximal rover footprint which are isolated and used to calculate geometric terrain metrics. Rover image courtesy of edupics.com.

work, so lower values are worse than higher ones, Figure 2.2.

- 'Slope' metric is defined by the best fit plane to the set of points, this metric is the angle between this planes normal and the local gravity vector. This metric is scaled based on the known maximum tipping angle of the rover and a safety factor.
- 'Roughness' metric is the largest distance between any point and this best fit plane, this is a measure of how 'un-flat' the terrain is. This metric is scaled by the ground clearance of the rover body and suspension and is used to avoid the risk of grounding on large rocks.
- 'Step' metric is the largest elevation difference between any two adjacent DEM cells, this metric is scaled by the maximum obstacle height that the rovers wheels are capable of driving over.

2.2.3 Archiecture of the Exomars Rosalind Franklin Rover

The Exomars rover will be the European Space Agencys first planetary rover mission, after many delays it is scheduled launch in the September 2022 Mars transfer window. The primary goal of the Exomars programme is the search for signs of Martian life, past or present. This includes the trace gas orbiter which will measure the chirality of methane in the upper Martian atmosphere to determine if its origin was geological or biological [85]. The primary science payload of the rover phase of this mission is a drill and the Pasteur biological instrument suite that will collect

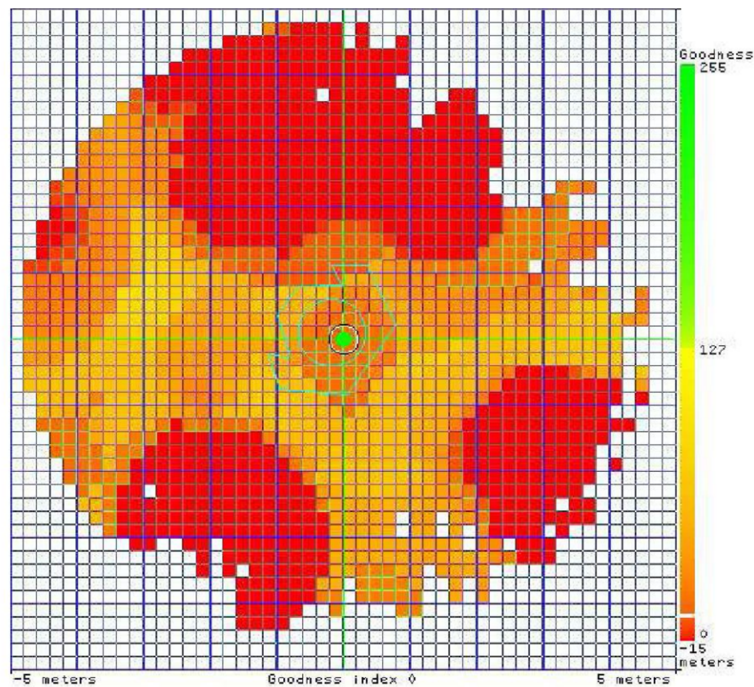


Figure 2.2: Goodness (Cost) map generated by Spirit on sol 107 of its mission. Red areas are impassible and yellow/green are traversable with differing levels of ease. Taken from Biesiadecki et al. [21]

samples from up to 2 metres below the surface and search for organic compounds [142]. The harsh radiation and photochemical environment of the top 50 cm of Martian regolith is expected to have destroyed any organic molecules if they existed there. It is hoped sampling beneath this level may find compounds that are unstable on the surface [141].

It is hoped that the Exomars rover will set new speed records on the Martian surface and push the boundaries of autonomous operation, these are ambitious goals requiring a novel design of GNC system to accomplish. Like the MSL and MER rovers before it Exomars will perceive its environment using multiple sets of stereo cameras [99] [129], and propel itself across the surface using a six wheeled locomotion system. Proprioceptive sensors are also present, an Inertial Measurement Unit (IMU) senses linear and angular acceleration while wheel odometry and bogie angle sensors report the motion and current draws of the locomotion system. Its On-board Computer (OBC) is comprised of two 96 MHz LEON2 processors one dedicated to data handling and the other to the autonomy system.

The rover has a hybrid autonomy system which at the top level uses a SPA paradigm to navigate its way across the surface. Every two metres the rover stops and performs the Sense and Plan cycles, using its navigation cameras to perceive the environment in front of the rover and generating a navigation cost-map, this map is then used to plan a safe route for the next two metres. The rover then follows this route in the 'Act' cycle with a reactive system, using both proprioceptive sensors and VO to monitor progress.

A pair of wide-band navigation cameras are mounted on the rovers mast, along with a pair of high resolution multi-spectral wide angle cameras. Either of these redundant stereo cameras can be used by the GNC system to generate elevations maps. These elevation maps are then converted into navigation cost-map and used to plan safe routes for the rover to drive. Due to the limitations of the depth precision, these maps and their associated routes will only be generated up to seven metres from the current rover location.

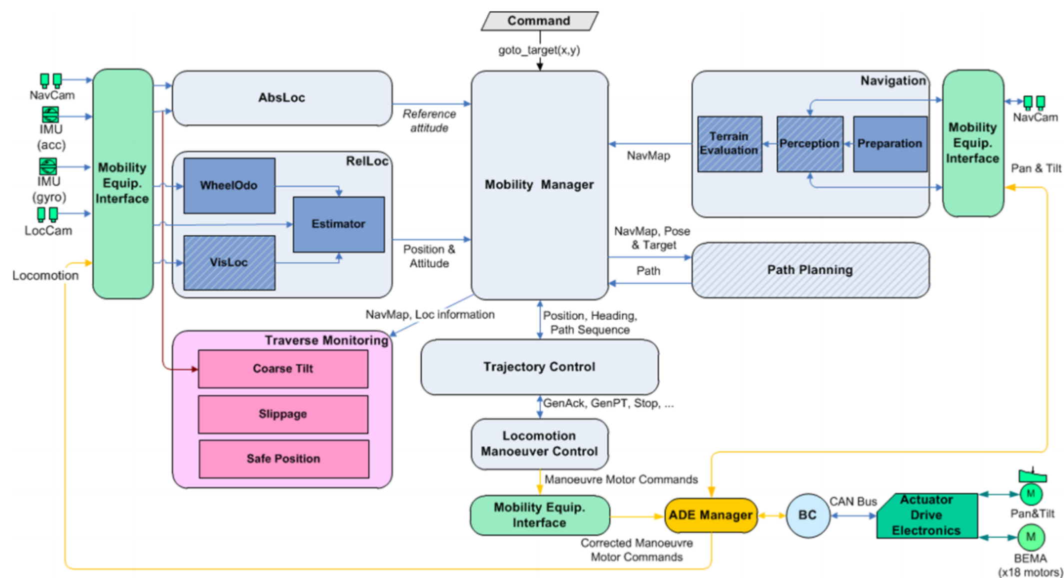


Figure 2.3: Architecture of the Exomars rover on-board GNC systems, courtesy of Airbus [154].

After a two metre route segment has been planned the GNC system switches to a reactive path following mode, both proprioceptive sensors and the exteroceptive localisation cameras are used to determine the relative position of the rover in this mode. Wheel odometry and inertial measurements are computationally easy to integrate so this pose is updated at around 100 Hz however this estimate is not robust enough to be used alone. Wheel slippage and sinkage cannot

be detected using these sensors and inertial measurements are prone to drift. To correct these sources of error a VO system is used, based upon stereopsis using a pair of hazard cameras mounted on the rovers belly which produce relative pose estimates once every 10 seconds [129]. In order to achieve this, relatively slow, update speed the only one in five cycles of this system perform full stereo matching the remaining four cycles use a single monocular image, its features being matched to 3D feature locations determined in the most recent stereo matching cycle.

Since VO updates are the only method for detecting slippage and sinkage this 0.1 Hz update rate is the limiting factor for the speed of the rover. In the worst case scenario this system can take 20 seconds before it will detect anomalous rover motion. This 20 second worst case combined with how far the rover could dig itself in over that time determines the maximum safe speed of the rover. It can be seen that processing algorithms and the hardware used in this part of the GNC must be as optimised as much as possible, given the effect they have on rover speed.

Exomars has smaller wheels and therefore higher wheel pressure than Curiosity, which was designed to be buoyant whilst traversing soft sand, to address the risk of sinkage whilst traversing over soft terrain Exomars uses a novel wheel walking approach [107]. On soft terrain the suspension deployment motors combine with the wheel motors to ‘walk’ three of its wheels forward while the other three are braked. Followed by the opposite three wheels doing the same, this significantly reduces the traction forces needing to be transmitted to the terrain reducing the risk of wheel slippage. As well as the mechanical advantages of the wheel-walking technique, forwards motion is slower in this mode meaning that the 0.1 Hz visual pose updates are geometrically closer together. This lower speed further reduces the risk of sinking too far before the VO system can detect that sinkage.

2.3 Rover Nagivation Sensors

Complex, ambitious sensors have been carried by planetary rovers such as MSL Curiosity’s Chem Cam instrument [103] or the WISDOM Ground Penetrating Radar (GPR) instrument [34] on ESAs Roslind Franklin rover. Such exotic sensors however are reserved for scientific payloads, exteroceptive navigation sensors used by GNC systems are almost universally solid-state optical cameras. GNC perception sensors are mission critical components, therefore

reliability is prioritised above all else. Heritage optical cameras meet this requirement since they have no moving parts, and decades of flight heritage with proven reliability. Additionally they are lightweight and consume a small amount of power.

These cameras can be divided into two sets: Hazard cameras are mounted on the rover body and used by reactive GNC processes for VO and obstacle detection [98]. Mast cameras are used by the SPA GNC process to plan the rovers route and localise itself relative to past maps and lower resolution orbital maps [106]. Mast cameras are optimised for 3D perception of terrain around the rover, being located as high as possible to shorten the occlusion shadows produced by rocks. Their stereo baseline is as large as possible to decrease depth estimation errors, 500 mm in the case of the Exomars Wide Angle Camera (WAC)s [36].

Mast camera images are used for 3D GNC perception as well as scientific data collection. Both Exomars and MSL Curiosity carry high quality multi-spectral stereo imagers as well as lower quality single channel wideband stereo imagers [36] [100]. This adds redundancy into the sensor suite and allows the most suitable sensor to be used at a given time. The Exomars GNC system is able to use higher quality WAC stereo pairs taken with the wideband red filter when time and power allows or wideband stereo images from the NavCam if time and power is more constrained.

Section 2.3.1 describes the implementation and performance of stereopsis on the current generation of Mars rovers, while Section 2.3.2 discusses the opportunities and challenges of using LIDAR sensors. It is concluded that high value rover missions including the Mars SFR baseline for our work, will use stereo cameras for 3D perception. It is expected that LIDAR will be adopted when this technology matures, and higher risk missions such as commercial lunar rovers could be where this sensor is proved.

2.3.1 Steroposis

Stereopsis is a biomimetic technique which uses the baseline distance between two cameras to infer depth information from images pairs. Along with depth of focus is it one of the two mechanisms by which humans and many other creatures perceive the world in three dimensions. This mechanism was first formalised and applied to photographs by Albrecht Meydenbauer in 1858 to record the 3D shape of buildings and cultural artefacts [3]. The earliest and still most

common 3D perception systems used in robotics is stereopsis, only in recent years has direct 3D perception using LIDAR and RADAR been adopted.

Depth information is extracted from image pairs by computing the horizontal disparity between common features in both images. Values in this disparity map are inversely proportional to the depth of the objective feature at that point [60]. Before this process can be performed each image must be un-distorted and projected to a common plane (rectification). This process means that a matching pair of horizontal pixel rows and the optical centres of each camera will always lie in a single plane, this is essential for disparity map generation.

Disparity map computation is analogous to autocorrelation in a spatial context, at each point along an epipolar line the disparity is found which minimises the difference between the pixels of the left and right images. If sufficient detail exists in both images then it is possible to compute these disparities with sub-pixel precision [109]. However this ideal situation does not always occur, if images contain little detail or smooth areas then disparity errors will increase or fail to be estimated at all. The relationship between disparity error and depth error is described in Equation 2.1, taken from Gallup et al [54].

$$E_z = \left(\frac{Z^2}{B \cdot F} \right) \cdot E_d \tag{2.1}$$

Where E_z is the depth estimation error, Z is the objective depth, B is the baseline between the optical centres of the cameras, F is the focal length, and E_d is the disparity estimation error. It can be seen that depth estimation error is quadratic with respect to depth, meaning that the effective range of stereo camera systems will always be limited by this factor. The remaining three terms can be controlled within reason to reduce this error. Increasing the baseline or focal length will both decrease depth errors, at the cost of reducing the overlap between left and right images. This overlap reduction reduces the effective Field of View (FOV) of the stereo camera system. Disparity estimation error E_d can be reduced using higher resolution sensors, or improved disparity algorithms but only to the limit of visual sharpness in the scene.

JPLs MSL Curiosity rover carries two redundant pairs of navigation cameras on its mast as well as eight redundant hazard cameras mounted on its body. These cameras are identical to those used successfully on the two MER rovers. These monochrome cameras have a spectral range of 600 - 800 nm optimised to perceive detail on the Martian surface. Mapping and global

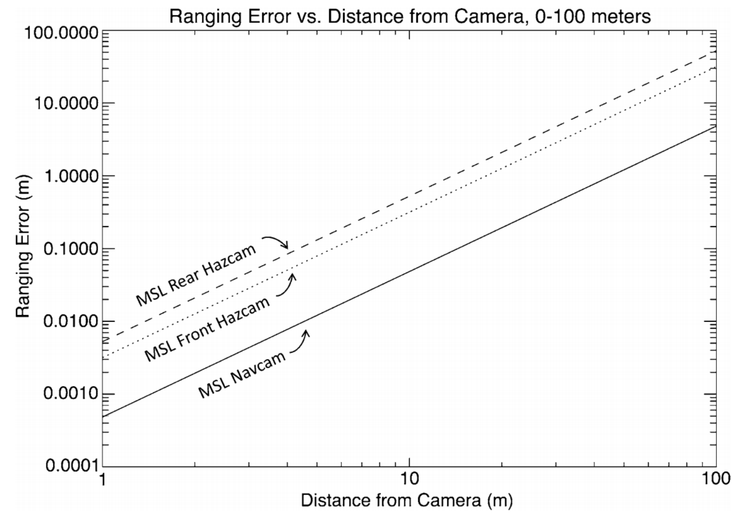


Figure 2.4: Stereo depth estimation errors of the MSL naviation and hazard cameras, taken from Maki et al [99].

localisation is performed using the mast mounted Navigation cameras, which have a resolution of 1024 x 1024, FOV of 45 degrees, and a stereo baseline of 422 mm [99]. The remote sensing mast these cameras are mounted on is capable of pointing at any part of the terrain around the rover, allowing it to produce panoramic maps if needed. The measured depth estimation error of the navigation cameras is shown in Figure 2.4 using an assumed disparity estimation error of 0.25 pixels.

The Exomars rover Rosalind Franklin collects mapping data using a pair of mast mounted cameras, again these are single channel cameras with a resolution of 1024 x 1024, a FOV of 65 degrees, and a stereo baseline of 150 mm [153] [129]. The on-board computers used on this rover are more restricted than the nuclear powered MSL rover, so a more optimised stereo pipeline has been developed. The navigation cameras developed by Neptec Design Group, include an Field Programmable Gate Array (FPGA) which performs un-distortion and rectification internally, transmitting the corrected images directly to the GNC computer. To speed up the disparity computation the near-field in the lower part of the image is generated at half the resolution of the top half [154]. Points in these nearer the rover will be denser due to their proximity.

Depth estimation error for the full and half resolution depth estimates are shown in Figure 2.5 alongside those of the MSL navigation camera for comparison. Full resolution Exomars

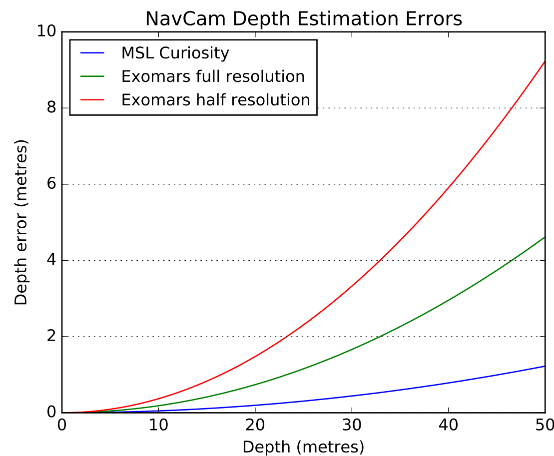


Figure 2.5: Comparison of navigation camera depth errors for MSL Curiosity and Exomars, full and half resolution disparity maps, based on published values from [99] [129].

navigation camera depth errors are approximately 3.8 times greater than those of MSL, this is mostly due to the larger baseline used on curiosity although Exomars' wider FOV also contributes. Alongside these theoretical 'best case' depth errors, even greater errors will occur on surfaces will little detail. This is particularly relevant on Mars where dune fields are common and hazardous to rovers.

2.3.2 Lidar Sensing

LIDAR sensors are commonly used on terrestrial autonomous ground vehicles, 3D information is produced directly by measuring the time that reflected light takes to travel between the sensor and an object. This is a marked difference from the inferred 3D information produced by stereopsis which has greater uncertainty and requires computationally expensive processing. LIDAR sensors have a key advantage over stereopsis, being an active sensor they are immune to changes in ambient lighting so function equally well in the dark as they do in extreme lighting such as sun set or sub rise.

Although not strictly LIDAR it is worth noting that JPLs sojourner rover used a structured light sensor [47] to improve the performance of its stereo cameras [124]. Five laser stripes were projected in front of the rover, allowing faster 3D reconstruction than was possible using stereopsis alone. It is notable that this type of sensor was dropped from future JPL rovers as the

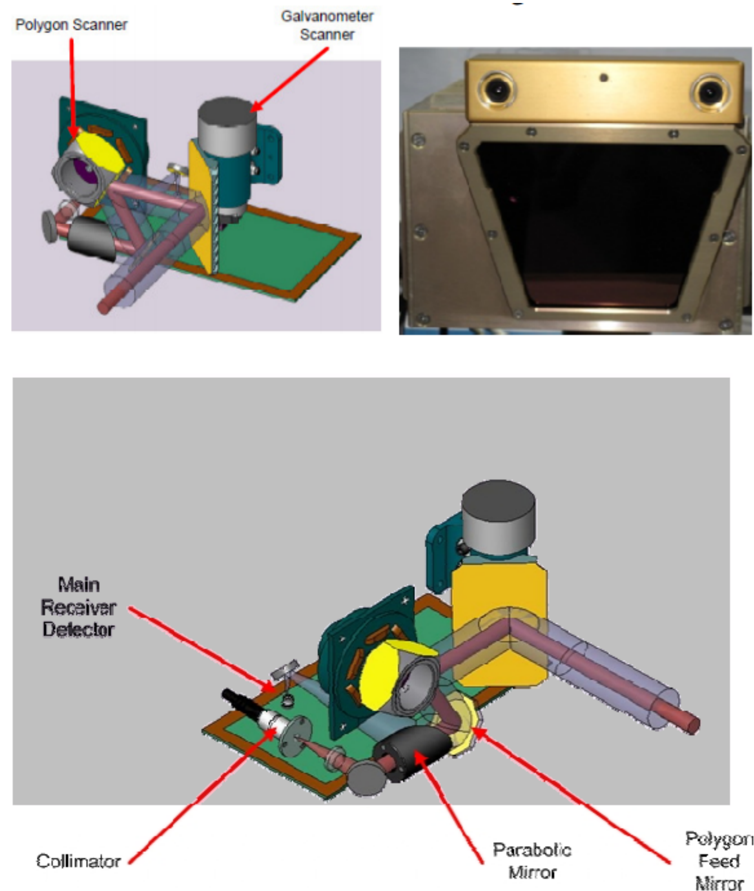


Figure 2.6: Opto-mechanical design of the fast scanning LIDAR developed by Bakambu et al. taken from [13].

power of on-board computers increased.

The advantages of LIDAR over stereopsis the cause of their widespread use in terrestrial robotics, use in space and particularly planetary rovers however has been restricted by the additional challenges of this environment. Measuring distances to cm precision by timing the movement of light requires electronic Time to Digital Converter (TDC)s operating at pico second scales [113]. Modern high frequency digital electronics is capable of meeting this requirement with relative easy, however these types of circuit are particularly susceptible to radiation damage, Section 2.5.2. Space qualified LIDAR sensors require less efficient analogue front-ends which in-turn require more powerful lasers to function. Scanning LIDAR sensor such as the fast scanning LIDAR developed by Macdonald, Dettwiler And Associates (MDA) [13] additional contain moving parts which need to operate at optical precision directing the laser and return signal

paths, Figure 2.6. Engineering these parts to survive launch and operate in vacuum, significantly increases their cost and mass.

The results of space qualifying these LIDAR sensors results in a mass of at least 5 Kg and power consumption in the region of 25 Watts, [13]. To date these sensor have only been used on orbital spacecraft while performing proximity operations during docking [121] [4] or close approaches to asteroids [152] [159]. These mass and power requirements are the primary reason why LIDAR sensors have not been used on any planetary rover missions to date, and why they are not under consideration for the future Mars SFR rover Airbus is developing.

2.4 Navigation Cost-map Generation

Early autonomous mobile robotics research was largely focussed on indoor built spaces where obstacles and surfaces are more predicable than outdoor environments. The maps estimated by these robots were occupancy maps representing only the presence or absence of obstacles [44]. Autonomous robots use these maps for two tasks, localisation of the robot, and planning of paths towards goals. While it is possibly to plan routes directly on an occupancy map this requires checking if any part of the robot will collide with an obstacle making the task computationally expensive. The concept of navigation cost-maps produced via obstacle expansion is now a standard solution to this problem [45], which allows efficient path planning using simple algorithms such as A* [64]. The shape of the robot is simplified to the smallest circle which encompasses its whole shape, then known obstacles on the occupancy map are expanded by the radius of this circle Figure 2.7. The resulting navigation map can then be used to plan paths as if the robot were a single point, greatly simplifying planning algorithms.

The concept of these navigation maps has been developed for more complex environments by describing the difficulty of traversing over a point as opposed to a binary possible/not possible value, Figure 2.8. These maps are suitable when mobile robots need to traverse unstructured terrain without prepared surfaces to drive on, and find use in agricultural robots [14], robotic search & rescue [31], and planetary rovers [67]. The precise meaning of these maps and methods used to generate them are more complex and application specific than expanded obstacle maps.

Navigation cost-maps represent the difficulty of traversing over terrain as a grid of scalar values.

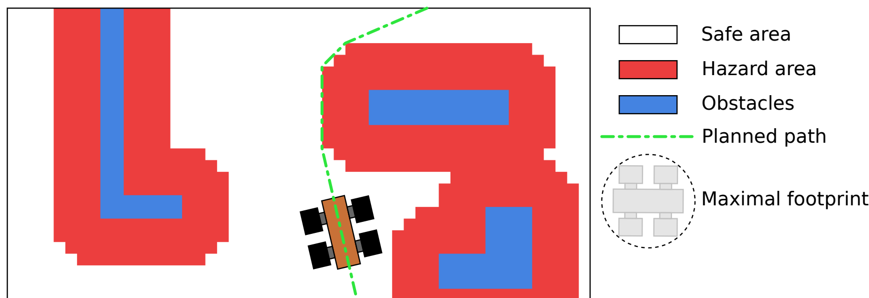


Figure 2.7: Example navigation map produced using obstacle expansion. Obstacles (blue) have been expanded by the radius of the maximal rover footprint into hazard areas (red), the resulting map can be used to easily plan safe routes by finding the shortest line which does not enter the hazard area (green).

In many cases these values are not strictly defined metrics, but simply values between zero and one, which capture to some degree the difficulty of driving over terrain [28]. This is sufficient for safe routes to be planned, although it does not guarantee that they will be optimal. It is possible to define cost-maps more formally to estimate quantifiable values, such as the time or energy required to drive over a cell [91]. These cost-maps can then be used to plan routes which minimise the relevant quantities.

Quantified cost-maps such as these are still an active area of research and practical generation techniques which accurately estimate them are still under development [139]. The trade-off between the complexity of the cost-mapping algorithm itself and the benefit of more optimal routes is not clear cut at this time. An interesting related area of research that should be noted, are planning algorithms which optimise the energy harvested on a route as well as the energy cost of locomotion. Solar powered rovers can use this technique to optimise routes based on both the terrain and the angle of solar illumination [125].

Although cost-maps are an established technique for mobile robot path planning, they are not without limitations. These maps simplify the cost of driving over a cell, in all possible directions, to a single value. This simplifications means they fail to capture the true cost of a path in many situations, such as the difference in slippage between up or down a slope as opposed to across it [127] or diving through narrow gaps[126]. Complexity and computational cost is the cause of these limitations, cost-maps can be extended into 3D where rover orientation is the third dimension, but memory and computational costs explode in this higher dimensional space.

2.4.1 Cost-map Generation Techniques

The simplest and most computationally efficient method to generation a navigation cost-map is to apply geometric measurements to the perceived terrain model, Figure 2.8. Properties relevant to the mobile robot in question are chosen such as mean slope and maximum step for wheeled rovers [21], or small-patch slope for quadruped robots. Existing geometric algorithms can be used to compute these metrics, which are then scaled based upon the capabilities of the robot in question [84]. The navigation cost estimates produce by this method are more approximate that more advanced methods, which requires them to deliberately over-estimate costs for safety reasons. Another limitation is the lack of surface material knowledge, all shapes are considered equal even if one is tarmac and another is sand.

Additional sensor information can be used to generate cost-maps which either directly measures or infers the material properties of terrain. Direct sensor measurements are used to classify the type of material, which is then used to estimate terra-mechanical properties [76]. These techniques have been researched for both terrestrial applications [97] and planetary rovers [67]. JPL no longer release technical details of the GNC systems they are flying on Mars rovers, but it is likely given the time of the work by Helmick et al [67] that this approach has been adopted by both the MSL Curiosity, and Mars 2020, Perseverance rovers.

The most precise cost-map generation algorithms are those which use a full dynamical model of the rover chassis, along with a terrain map including terra-mechanical estimates. Exomars uses a form of this technique to estimate the wheel pressure that would be applied on the terrain map. More complex algorithms can simulate the dynamics of the rover chassis moving over terrain to generate the most precise estimates [117]. Both the computational cost and the detail configuration required to use these algorithms mean they are not commonly used, in the case of Exomars a highly simplified algorithm was distilled in order for it to execute on the radiation hardened processor used.

The generation of navigation cost maps using Deep Learning (DL) is a less mature field, the earliest published work the author is aware of is Wulfmeier et al in 2016 [155]. Many application specific solutions have been proposed, which is not hugely surprising given that cost map functions vary significantly between different domains: An autonomous car, agricultural tractor, or planetary rover can all use cost maps but the cost functions used will vary greatly.

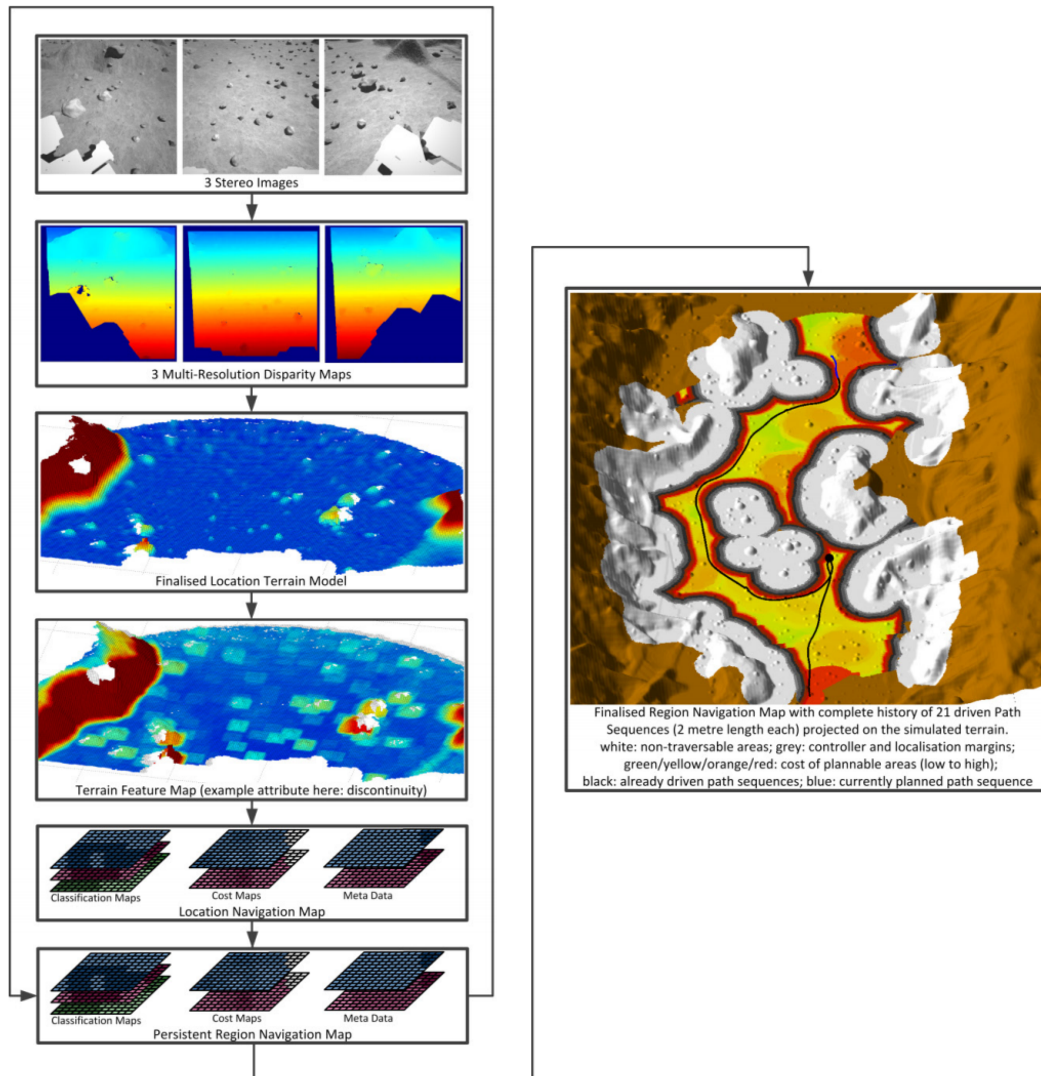


Figure 2.8: Exomars on-board mapping perception pipeline which captures pan cam stereo pairs and processed them into navigation route plans towards the requested goal. Courtesy of Airbus [154].

Wulfmeier et al [156] demonstrated a Reinforcement Learning (RL) approach to generating cost maps for autonomous road vehicles using data fused from a range of sensor types. Wei el al [150] use an encoder-decoder model to estimate cost maps from registered aerial images. Drews et al [43] used a Recurrent Neural Network (RNN) to generate cost maps directly from an input video stream. Although none of these models are a perfect fit for the cost mapping task on a planetary rover, they demonstrate that the concept has been successfully tackled before.

2.5 Machine Learning Onboard Spacecraft

Machine learning has seen considerable use in the space industry to date, however the majority of these applications are within the ground segment. Commercial processing of Earth Observation (EO) datasets is now routinely facilitated using ML models to improve metrology [89], segment cloud cover [11], and perform sensor fusion [143] amongst many others. However these applications are far removed from the operational challenges of space, and are performed without posing any risk to the space-borne observatories where this data originates.

Two significant challenges have to be addressed before ML is able to safely perform mission critical functions on-board the majority of spacecraft. ML models must be accommodated within the limited computing power of radiation hardened CPUs, and functional V&V of ML solutions to the levels required by space industry will need to be addressed [33]. ML models are traditionally computationally intensive solutions, usually executed on GPUs. however the computing power available on-board spacecraft is far more limited than it is on terrestrial computers and space qualified GPUs do not yet exist. Mass, electrical power, and heat dissipation are expensive and finite resources [115]. Then there is the harsh radiation environment of space which increases in severity beyond Low Earth Orbit (LEO) and for longer duration missions, as discussed in Section 2.5.2.

The limitations of fully space qualified CPUs can be avoided somewhat by missions in LEO. These orbits, below 1000 km in altitude, enjoy significant radiation shielding by the Earth's magnetic field. There are patches of higher radiation such as the South Atlantic Anomaly (SAA) [66] with increased high energy electron flux and over the poles where Galactic Cosmic Rays (GCR)s increase [17]. Overall though LEO has a relatively benign radiation environment

when compared with higher orbits and deep space, allowing Commercial of the Shelf (COTS) processors to be flown successfully on numerous missions [130] [146].

These COTS processors have been used to demonstrate ML image processing on-board LEO spacecraft. Manning et al demonstrated a satellite which used a COTS Zynq chip [157] to process EO images on-board [101] [55]. This enabled the satellite to automatically select the most suitable images is for downlinking, making more efficient use of its ground station contacts. However these ML models are limited to data handling tasks to avoiding the need for strict V&V, and are too computationally intensive for the radiation hardened computers necessary deeper into space.

ML techniques with lower computational cost have been investigated for missions beyond LEO Castano et al [29] studied the use of Support Vector Machine (SVM)s on-board the Mars Odessy spacecraft to perform on-board estimates of atmospheric dust and ice distributions using THEMIS [105] data. This technique was evaluated theoretically on datasets, as well as being deployed onto a flight representative computer. The trained SVM was hand implemented in C then executed on a PPC750 testbed which emulated the RAD 750 processor [18] used on-board this spacecraft. The scale of the two SVMs was reduced by over an order of magnitude to 40 support vectors each to achieve the required performance on this processor.

The applications studied by Manning et al and Castano et al both optimise the use of space vehicle downlinks by autonomously selecting salient observations on-board. In the Mars Odessy case the ML solution was deployed to a fully radiation hardened computer, by minimising the implementation as much as possible. Both of these ML algorithms perform payload data processing tasks, so their V&V requirements are not as strict as that of critical spacecraft software.

Autonomous control of critical spacecraft functions has been a demonstrated on range of flight missions such as ESAs Project for On-Board Autonomy (PROBA) and rosetta [19] [46] and NASAs OSIRIS-Rex [96] [20] to name but a few. None of these missions however are using DL models in their decision making processes. Resources are allocated using dynamic algorithms, and pre-planned tasks are executed using autonomous perception and safety systems.

Missions have made use of ML and on-board autonomy, but to date none have used ML in an autonomous system. Two main obstacles are preventing the advantages of ML being used in

deep space missions; the limitations of radiation hardened processors, Section 2.5.2 and the difficulties of integration them into the flight software development process, Section 2.5.1. The following sections explore these challenges in more detail and survey the current state of ML deployment tools and their suitability for use on a high value space mission.

2.5.1 Software Development and Validation & Verification

The challenges of adopting ML solutions to critical spacecraft operations has been discussed in Section 2.5. Fundamentally these same challenges are faced by any flight control software, and the route through their development up to a flight mission is described by flight software development standards. The software development requirements and V&V processes of flight software comprise a wide and complex discipline. This section presents a brief overview of these standards and requirements. European Cooperation for Space Standardization (ECSS) define the flight software processes that are required by high value ESA missions and can be found in the ECSS-E-ST-40C (software engineering) [48] and the ECSS-Q-ST-80C (software product assurance) [49] standards.

Due to the strict software engineering requirements imposed by ESA and NASA on high value missions, final flight software for high value missions is commonly written by specialist contractors. These companies have access to the skills and tools required to perform the stringent verification which is necessary. This was the case for the Exomars rover, Rosalind Franklin, Airbus developed prototype GNC software for the LEON 2, which met performance and operational requirements. This software was then passed to an external contractor who re-implemented and verified the GNC software stack to ESA standards.

In order for this process to be possible access to the complete code base including any non-standard library dependencies is needed. Even if the source code of external libraries is available it is preferable avoid them due the time and cost of implementing them in to flight standards. It is not always feasible to re-implement algorithms to ESA standards due to the restrictions placed on code. Dynamic memory allocation is a potential source of many bug types, especially when systems are required to run for long periods of time. For these reasons dynamic memory allocation is not permitted in any flight software [71].

For a range of reasons flight software is required to be written in ANSI C. The heritage required

for the development tools is one reason behind this, compilers, linkers and debuggers which have multiple decades of use are available [71]. These are not the most efficient or easiest tools to work with but the level of confidence in their correctness is very high. The nature of the C language itself is also better suited to the tight resource constraints of flight software than more complex languages such as C++. While this is a powerful language, binaries are easily bloated by polymorphic objects and template definitions [144].

Space V&V practices are concerned with ensuring that flight software is both functionally correct and safe to execute [147]. Functional correctness is the requirement that algorithms produce the output/behaviour that is required. Code is considered safe to execute if it can be ensured to complete executing within a certain duration of time and not use more than a known amount of RAM. These limits will take into account maximum utilisation ratios of at most 50% for critical flight software [110].

Analytically proving that the output of a ML model will meet a set of requirements is still an open research question, with a much work still to be done. The field of ML introspection is still a young topic with concepts such as natural language [15] and input salience [94] being developed to try and understand how ML models are affected by their inputs. A more familiar verification technique to the space industry is testing, a large test set which covers edge cases sufficiently well has been suggested by our sponsor Airbus as an acceptable method for use with ML models.

Unlike the functional verification of ML models which is more challenging than conventional algorithms, execution safety analysis is more straight forward. Tensor flow graphs which are used to execute ML models are sequences of fixed size tensor operations. It is possible to have flow control operations within models but this is not common and is not used by any of the models presented in this work. Therefore each of the cost-mapping models presented will have a fixed execution time and RAM requirement, which is not affected by the values of the input tensor.

2.5.2 Computational Power Limitations

The first computer to be constructed from silicon microchips was the Apollo guidance computer [82], it is ironic therefore that space qualified microprocessors today lag so far behind their

terrestrial counterparts. This disparity is primarily caused by the shrinking size of silicon transistors that can be fabricated resulting in their increased sensitivity to radiation effects. The use of specially designed chips which mitigate these radiation effects and the low volumes they are produced in, is why the computers used in deep space are so much slower and more expensive than their earthbound counterparts.

A secondary reason is the strict reliability requirements of value deep space missions. These requirements necessitate the use of thoroughly tested systems with long heritages of success. There is a risk that new processor designs may contain design flaws, even with the rigorous testing and evaluation they go through. Such flaws have been known to be discovered after chips have been in production and use for time years. A classic example is the Intel Pentium Floating Point Divide (FDIV) bug discovered in 1994 after the CPU had been in production for over a year [118]. These two reasons mean that legacy CPU architectures with long heritage are used along with radiation hardened fabrication processes and fault tolerant design methods, resulting in performances which lag significantly behind that of cutting edge terrestrial processors.

There is no easy mechanism to access more CPU power on deep space missions: If higher risk is accepted then the latest generation of radiation hardened processors can be used. If more of the power and mass budgets is used for On-board Data Handling (OBDH) then more computers heritage computers can be used. Moving away from CPUs however challenging algorithms are often deployed to FPGAs on-board spacecraft, and this technique holds great promise for implementing ML. This is still an active research topic with no automatic tools deployment tools available [145] [162]. Although this is a potential solution to the deployment of ML models on radiation hardened hardware, it comes with significant development and V&V challenges.

Machine Learning is a notoriously computationally expensive technique to deploy, most terrestrial applications are cloud based and GPUs are used to execute models on mobile robotic platforms. Recent work however has started to find utility in smaller ML models which can run on embedded computers and micro-controllers while still performing useful tasks such as person detection [116]. If utility can be found for such smaller models for space applications then the computational limitations in deep space may be avoided.

This section will present a brief description of the effects of radiation on microchips and the techniques used to harden them. Currently available radiation hardened processors will then

be surveyed. The computational performances listed here will be used to assess the feasibility of ML models proposed later in this work. It will be shown that the current state of the art processors are capable of 200 to 400 Million of Instructions per Second (MIPS) while the best next generation chips, which exist but lack the same level of space heritage are capable of over 5000 MIPS.

Radiation Hardening Challenges and Solutions

Radiation effects Complimentary Metal Oxide Silicon (CMOS) microchips in two ways: Single event effects are caused by a single particle strike and usually cause temporary upsets to the logic levels in a circuit but in some cases can cause permanent damage. Total Ionising Dose (TID) damage is caused by particle strikes cumulatively depositing charge into the insulating layer of CMOS transistors, over time this increases the charge needed to open or close the transistor [40]. The effect is a gradual increases the leakage current of the device until the charge builds up to a point that transistors are unable to switch. When this point is reached, nothing can be done to reverse the damage and the device has reached the end of its operational life. For this reason TID tolerance is especially important to long duration deep space missions, which expect to experience high levels of radiation [63]. These expected doses include the unlikely but unpredictable case of a large solar particle event occurring during a mission.

Single event effects fall into three classes [112]: Single Event Upset (SEU) such as a memory or latch flipping between states, Single Event Transient (SET) such as a signal trace being charged to an incorrect logic level and Single Event Latchup (SEL) semi-permanent events which can lock a memory register into a fixed state that can only be fixed by power cycling the device. Finally Single Event Burnout (SEB) are a special destructive case of SEL where a particle switches a transistor such that a short circuit is created. Unless these are rectified within milliseconds, these events can permanently destroy a circuit within a chip.

These single event effects can be mitigated through a range of radiation hardening by design techniques, a full discussion of this area is beyond the scope of this thesis, but they are outlined here so their effect on CPU performance can be understood. SEUs can be protected against using error correcting codes within both off-chip memories and on-chip data paths, these extra checks add delays in accessing memory and between blocks on the data path [80]. SET events are harder

to detect but can be protected against using multiple execution. Latch up and burn out resistance requires that chips are designed using special radiation hardened transistor layouts. Radiation hardened component layout libraries have been developed by several chip manufacturers [131] to avoid these faults. While these libraries do not increase the component count of a chip, the die area used and power consumed is higher than normal libraries which contributes to the slower clock speed of equivalent radiation hardened designs.

Review of Available Radiation Hardened CPUs

- IBM RAD6000

  Designed in the late 1990s the RAD6000 radiation hardened single board computer went on to be used on many deep space missions including NASAs MER rovers Sprit and Opportunity [73] and the Pathfinder lander [59]. This CPU and single board computer has a TID tolerance of 100 Krads and is available at clock speeds up to 33 MHz across its full operating temperature range, at this speed the processor is capable of 35 MIPS and 26 Mega Floating Point Operations per Second (FLOPS) [59].

- BAE RAD750

  The successor to the RAD6000, now being produced by BAE systems was first flown in 2005 on the Deep Impact comet mission [86], with a clock speed of up to 200 MHz and a performance of 400 MIPS, Figure 2.9. This system has a TID tolerance not less than 100 Krad again making it suitable for long duration deep space missions [18]. Notably this is the CPU used on the MSL rover Curiosity, on this rover the CPU is shared by the autonomy systems and on-board data handling system so less than 75% of CPU time was available for the autonomous systems [12].

- LEON 2

  The LEON 2 processor was developed by Gaisler Aeroflex for the ESA and radiation hardened versions (AT697, AT7913) are currently available from Atmel, these 32 bit processors are available with clock speeds up to 100 MHz and are capable of 86 MIPS and 23 Mega FLOPS [7]. Two LEON 2 processors will be used on the ESAs upcoming Exomars Rosalind Franklin rover, one of these processors will be dedicated to housekeep-

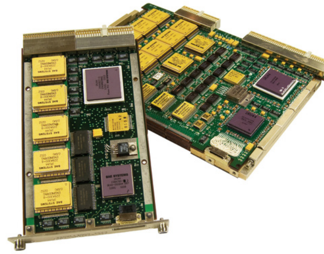


Figure 2.9: RAD750 single board computer currently in use by the MSL curiosity rover on Mars, courtesy of BAE systems.

ing and data handling, allowing the second processor to be dedicated to the perception and autonomy systems.

- LEON 3

  The GR712 from Cobham Gaisler is the most powerful LEON 3 radiation hardened processor with space heritage, it is a dual core LEON 3 CPU with a clock speed of up to 200 MHz [2], Figure 2.10. At this maximum clock speed the processor is capable of 200 Mega FLOPS, a significant increase from the LEON 2 family of processors which preceded it. Due to its space heritage this processor is likely to be the used for the next SFR rover, for this reason it is the baseline processor for testing if the techniques proposed in this work are feasible for near term planetary rover design.

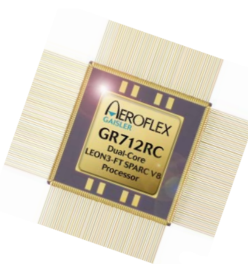


Figure 2.10: The GR712RC radiation hardened space processor, courtesy of Gaisler Aeroflex.

- LEON 4

  Recently completing development, with engineering and flight units now available is the Cobham Gaisler GR740 next generation of European high performance space processor, Figure 2.11. It is a quad core CPU with a minimum clock speed of 250 MHz across its

full operating temperature range (-40 125 o C) and is capable of up to 1000 MIPS [52]. It will be some time before this processor builds up enough heritage to be considered for flagship rover missions such as Exomars, but it clearly shows the amount of processing power that will be available to such missions being planned over the next ten years.

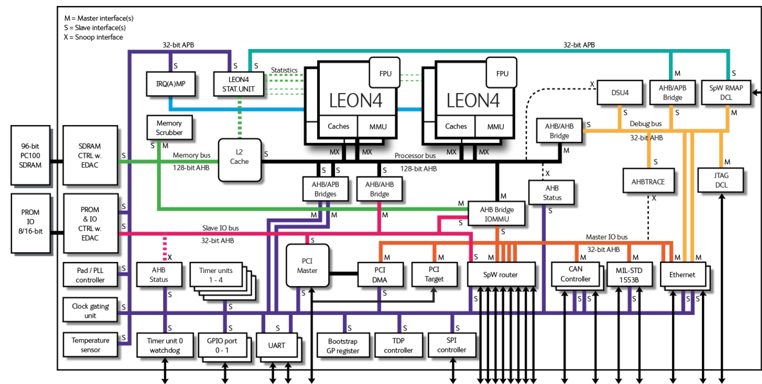


Figure 2.11: Block diagram of the GR740 quad core LEON 4 radiation hardened microprocessor. Courtesy of Gaisler Aeroflex.

- BAE RAD5500

  The successor to the RAD750, this is the first 64bit radiation hardened space processor and is available in single core (RAD5510, RAD5515) and quad core (RAD5545) versions, Figure 2.12. Although this processor lacks the level of space heritage of the RAD750 or LEON 3 families its performance is a significant step up, the quad core RAD5545 is capable of 5600 MIPS and 3.7 Giga FLOPS while consuming 20 Watts of power [134]. The performance level of this processor enables for the first time makes many computationally intensive machine vision techniques and large CNNs feasible in the context of a deep space mission.

2.5.3 Deployment Tools

A wide field of ML development frameworks have sprung up in recent times and these tools have been instrumental in the rapid progress of research which have been achieved. The majority of these frameworks include built in tools for the deployment of models onto targets ranging from full Tensor Processing Unit (TPU) pods on the cloud to hand held consumer devices. Here

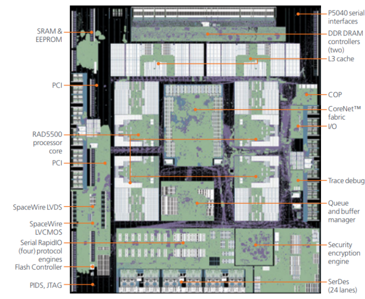


Figure 2.12: Labelled chip plot of the radiation hardened RAD5545 quad core space processor, courtesy of BAE Systems.

we review these deployment tools against the requirements of the flight software development process described in Section 2.5.1.

Tensorflow and the Keras tool which is built upon it remain the most popular ML research frameworks [1] [58]. Backed by Google this open source project started life as proprietary system at Google Brain and has enjoyed a large user base since its V1.0 release in 2017. three deployment tools are built into the Tensorflow (TF) framework: Tensorflow Lite (TFL) is aimed at mobile phone scale targets. The Accelerated Linear Algebra (XLA) compiler is used internally by TF to accelerate desktop CPU implementations through Just in Time (JIT) compilation, but is also capable of compiling models to bare-bones CPU targets. While Tensorflow Lite Micro (TFLμ) is aimed at embedded micro-controller scale targets. The latter of these tools it should be noted had not been released when our work started, so it will not be described here.

Core TFL is implemented in C++ 11 although Java support has since been added, GPU devices are supported allowing ML models to be accelerated even on mobile devices. Models are serialised to and from protocol buffers which store their structure and weights in a single definition. When the C++ Application Programming Interface (API) is used a pre built run-time library must be loaded which de-serialises the model allocates resources and executes the flow graph. Comparing this technique to the requirements described in Section 2.5.1 it is clear they are not met. The binary size of TFLs run-time is over 100 MB and is therefore an impractically large amount of code re-implement. The GPL license of this code is also problematic, derivative work is required to be released under the same open source license. This means that very expensive flight quality code would have to be freely released to the public, which would not

only be bad for business but could potentially breach export regulations.

It is clear that the TFL tool is not suitable for integration the flight software development process, even before its C++ compiler requirement and lack of support for any real time Operation System (OS) is taken into account. The XLA Ahead of Time (AOT) compiler on the other hand is more promising. This tool compiles ML models directly into binary code, removing the need for the large run-time of TFL, a lightweight support is required to execute these binaries. The versatile Low Level Virtual Machine (LLVM) compiler architecture allows new target CPUs to be used with only the addition of a new backed. It is also possible to export LLVM Intermediate Representation (IR) instead of binary code, in theory this allows any LLVM compiler to deploy a model via this tool.

The TF XLA compiler appears to be a promising option, with fewer technical challenges than TFL. However there are still problems with integration into the flight software development process. It does not generate prototype code beyond low level IR, which cannot then be manually re-coded to ESA flight standards. If it is considered as a compiler in its own right then it lacks the heritage and reliability expected of the tools usually used to build flight C code. In conclusion neither of the TF deployment tools which existed in 2017 are suitable for deploying ML models as part of a critical flight system, the XLA compiler however does appear promising for the purposes of deploying prototype models to radiation hardened CPUs.

Pytorch is another popular ML framework backed by Facebook, it is a more recent tool than TF and provides an API using an imperative 'code as a model‘ paradigm designed to make research tasks as efficient and easy as possible [114]. A wide range of containerised deployment tools are provided by Pytorch to execute models on cloud services such as Kubernetes [69]. Alongside these tools the TorchScript module allows models to be exported and executed in native C++ 14 projects [38].

The architecture of deployed TorchScript is close to that used by TFL, models are serialised to the file system and a large run-time is linked into the target binary which loads and interprets the model to perform inference. These similarities pose the same obstacles to use on radiation hardened CPUs and integration with flight software development processes. Being designed for use within a high-level OS means that significant re-designing would be needed for it to work with the more restricted real-time OS used for critical flight systems. The size of the run-time

library and its C++ 14 requirement put this tool out of contention.

In conclusion none of the deployment tools provided within the leading ML machine learning frameworks are suitable for integration within the flight software development processes necessary for mission critical system. There are both technical and procedural barriers to the use of these tools. Desktop software has long since moved away from the C programming language in favour of the more powerful and scalable C++ language. The use of C today is restricted to embedded micro-controllers, Digital Signal Processor (DSP)s and niche high reliability applications such as medicine, nuclear, and aerospace. There is little reason why these tools would target the C language given how much more restricted it is and its smaller user base. With the exception of TF XLA these ML deployment tools all target more powerful CPUs than the current generation of space processors, preventing their use. The use of high-level OS features not present on embedded real-time OSs is a further technical barrier. The scale of the deployment frameworks and ML models themselves will need to be reduced before they can be automatically deployed within mission critical flight software.

2.6 Gaps in Knowledge

Research into a new application for ML models has recently developed, the production of navigation cost-maps for mobile robots. This task is currently a performance limiting algorithm on-board planetary rovers operating on Mars and planned in the near future. Literature describing the existing cost-map generation algorithms has be reviewed along with the sensors and perception systems which feed into this task. Two gaps in literature have been found, regarding the suitability of ML cost-map generation for planetary rovers. The suitability of ML for this task in terms of its accuracy and computational requirements has not been studies, and a process for deploying ML models within the flight software development processor does not currently exist. These open questions and the best answers currently available listed in Table 2.1.

Section 2.4.1 reviewed current algorithms used to generate navigation cost-maps on-board Mars rovers and reviews the latest research into the new field of ML cost-map generation techniques. This process currently consumes a significant amount of time during the GNC loop, ML techniques have been demonstrated which can perform this task on terrestrial robots,

although this is still a new research topic with the earliest publication released in 2016. The authors are not aware on any work applying this technique to planetary robotics, given the nascent state of ML technique on Earth we feel it could bring novel benefits to this domain.

Section 2.5 has surveyed the challenges of deploying ML models on board the computers necessary in deep. Covering both the technical challenges caused by limited computing power, Section 2.5.2, and the process challenges of working within the software development systems required on high value missions, Section 2.5.1.

Table 2.1: Opportunities identified during this literature review.

Challenge	**Solution**
Is it possible to generate planetary rover cost-maps using ML?	• Research into use on terrestrial robotics is promising, no studies for planetary robotics currently exist.
Are ML cost-mapping models feasible to use on radiation hardened LEON3 computers?	• Performance known to be challenging on space processors, but no systems deployed and measured in literature.
Does a deployment process exist to implement this model within the flight software development process?	• Existing automated tools reviewed, none found to be suitable, hand coded implementation only option.

Chapter 3

Industrial Problem & ML Solution

3.1 Introduction

The goal of this research is to investigate an opportunity to use deep learning to improve the GNC capabilities of planetary rovers. The sub-system selected for this work is the cost-mapping process on-board a planetary rover. There are two mechanism for any replacement algorithm to improve a system: It could produce output superior to an existing algorithm, such as a more optimal route or longer range reconstruction. Or it could produce equivalent output to an existing algorithm while requiring lower computational resources. It should be noted that the latter case is only beneficial if the process in question is resource constrained. Luckily many processes on planetary rover GNC systems are resource constrained due to the heritage radiation hardened processors that must be used.

The cost-mapping task takes a map generated directly from sensor data, and produces a new map which estimates the difficulty of driving the rover over each cell of the map. The exact nature of input maps can vary but the output must be a map of scalar values that can be used by route planning algorithms. Two reasons lie behind this choice of application, it currently has a high computational cost, and superficially appears to be a good fit for existing ML models.

The two most computationally expensive navigation tasks performed by the current generation of planetary rovers are stereo disparity calculation and cost map generation [154]. The amount of time taken to perform these tasks puts them on the critical path of the GNC system, meaning

a speed up of these algorithms directly translates to an increase in the speed that the rover can cover ground. This in turn impacts the distance the rover can travel and therefore the number of science targets which can be studied.

Using deep learning to improve the performance of stereo disparity matching has been an active area of research for some time, with techniques focussed on end-to-end stereo matching [104] [42] as well as parts of the stereo pipeline such as feature matching [161] [56]. This research area is focussed on improving the results of stereo matching on platforms with significant processing power, ”Moreover, end-to-end stereo matching networks-based approaches basically require huge memory and are relatively time consuming.” Zhou et al. [163]. These algorithms can be utilised by autonomous cars and larger terrestrial mobile robots, which are capable of carrying and powering large computers. Executing these models in real time requires large (2020 context) GPUs so are far too large to be practical on conventional CPUs let alone radiation hardened processors.

The generation of navigation cost-maps using ML has been discussed in Section 2.4.1 which concluded that the concept itself was feasible, although no work to-date has been done on the planetary rover application. Given that all research into this technique described in literature has focussed on terrestrial applications, the computational cost and it’s applicability to radiation hardened CPUs will need careful analysis.

Other GNC tasks could have been considered such as the route planning algorithm used to generate paths across terrain. However these implementations do not take a significant fraction of the ‘sense’ and ‘plan’ phases of the GNC loop and the output of these algorithms is already optimal. Even a large increase in the speed of these algorithms would not have an impact on the duration of the ‘sense’ and ‘plan’ phase. There is therefore little value in for performance improvements to these algorithms.

ML appears to be a good fit for this task, since the input map (an $M \times N \times C$ tensor) is being converted to a cost-map (an $M \times N$ tensor). There is similarity between tasks already tackled by ML such as depth from monocular images [50], and auto-encoders [70]. Baseline cost-mapping algorithms already exist enabling supervised learning to be performed.

Our research into this application starts with the analysis of two standard types of ML model, to determine if they are able to perform the cost-mapping task. This is the most fundamental

question which must be answered before the more detailed application of this technique can be studied. Latter stages of this work studies two interrelated questions, can these ML models be executed on-board a planetary rover fast enough to be practical, and how can they be deployed within the software engineering processes used in the space industry.

The work described in this chapter has been published in the following conference paper:

- Blacker, P., Bridges, C.P. and Hadfield, S., 2019, July. Rapid Prototyping of Deep Learning Models on Radiation Hardened CPUs. In 2019 NASA/ESA Conference on Adaptive Hardware and Systems (AHS) (pp. 25-32). IEEE.

3.2 Industrial Problem Definition

Our challenge is to investigate the feasibility of producing navigation cost-maps (Figure 3.1) on-board a Mars rover using ML. In this case the input map is a single channel map of elevation values (a DEM), although the impact of other types of map are discussed in Section 3.6.1. A published cost-map generation algorithm used within JPLs GESTALT sofware [21] will be used to generate datasets for supervised learning. Emulated and engineering models of the LEON3 radiation hardened processor will be used to test the deployment process and evaluate the execution time of the proposed models.

As with any supervised learning task, sets of training data will need to be created and labelled, for this we were given access to two proprietary datasets as well as access to the Stevenage Mars yard to collect our own data. Labels were generated using the published GESTALT algorithm described below.

The reliability of the proposed ML solution will be evaluated by measuring its percentage error to three sigma, as described in Section 3.3.2. The computational requirements of models will be investigated to determine if they are feasible to execute on the radiation hardened CPUs necessary for Mars rovers. The target performance given for the cost-mapping task is to be able to process a map with an area of $77m^2$ and a DEM cell size of $4 \times 4cm$ into a cost map in 20 seconds. This area requirement is due to the 7 metre radius semi-circular DEM maps produced by the Exomars pan cam [154], during its ’sense‘ phase. This translates into a rate of 2405

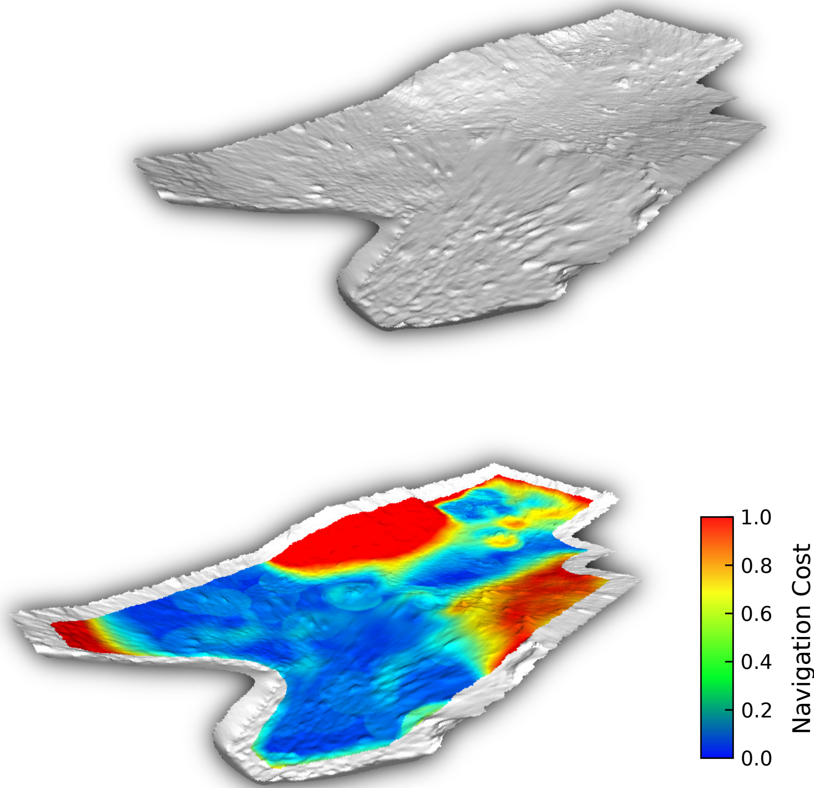


Figure 3.1: (top) Digital elevation map taken from the Airbus Marsyard. (bottom) Generated navigation cost-map of values, white border areas are of unknown cost.

DEM cells per second, and must be performed on a single core of a 200 MHz LEON3 radiation hardened CPU. Finally the challenges associated with deploying a ML algorithm within the software and V&V processes used on high value space missions will be discussed and potential opportunities described.

3.2.1 Existing Techniques

GESTALT is the full GNC system developed by JPL for use on their MER rovers Spirit and Opportunity, and is described in detail in Section 2.2.2. This GNC system uses a geometric approach to terrain assessment, resulting in three metrics. The maximum, slope, step and roughness of the surface at each location. These simple geometric measures are then scaled with the known capabilities of the rover chassis to measure the safety of driving over each area of the

map.

The exact cost mapping algorithm used on the Exomars rover is proprietary, and this information has not been shared with the authors, however some details are known about its requirements. Similar to the MER rovers, Exomars has solar wings protruding a significant distance beyond its wheel base. Potential collisions of these panels with terrain is an additional risk which the cost mapping algorithm needs to account for in addition to the usual locomotion challenge. The dynamical model of the rovers chassis is taken into account when generating the cost map to accurately reflect the behaviour of the rover [154], this is in contrast the the GESTALT system which used an entirely geometric approach with tuning weights. This technique has the advantage of producing more accurate navigation cost-maps but with an increase in computational cost.

3.3 ML Model Analysis

The goal of this section is to evaluate the accuracy of a range of ML models when used to estimate navigability. Our initial work published in [22] used CNN regression models to estimate individual cost map elements, the performance of these models was unacceptably slow compared to existing algorithms. Later work investigated Encoder Decoder models which are able to estimate multiple cost map cells in a single inference pass and displayed significantly improved performance. These ML models are described in Section 3.3.1, and results of both classes of model are shown so that the progression of this work can be discussed. Figure 3.2 describes the setup that was used to evaluate all the models used in this work.

In order to train and evaluate an ML regression model a set of inputs and expected outputs is required. The input to the navigability assessment task is a DEM which would normally be generated from upstream perception processes on the vehicle, for our experiment we use a set of four planetary DEMs captured using a range of sensor modalities. The variation in modalities is primarily out of necessity, these are not commonly available data sets so we have had to use those we have access to, however it does give us the opportunity to study the effect of different sensors on the whole mapping process. These datasets are described in more detail in Section 3.3.3.

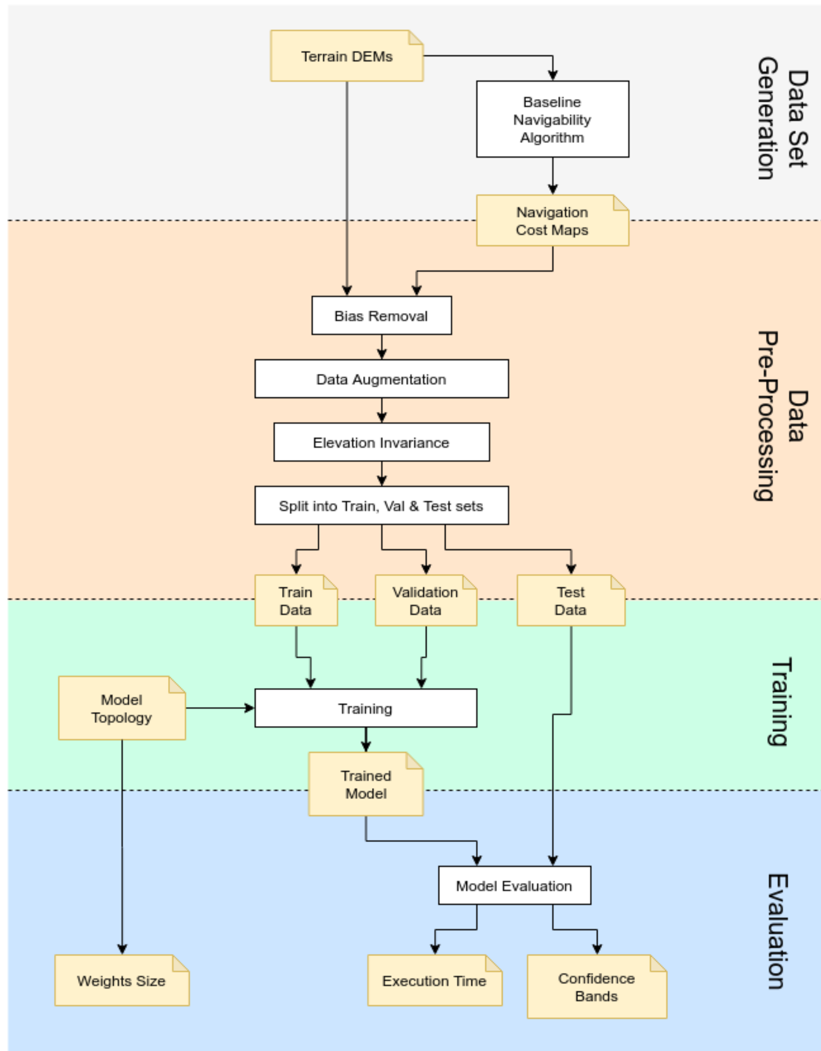


Figure 3.2: High level processing steps of our ML terrain estimator evaluation experiment.

The desired output of these models is a map of navigation costs over the terrain in question, this map should be closely correlated with the time or energy cost of driving a real rover over the terrain. Collecting this data directly using an actual rover is impractical given the quantity of training data required, so instead a published navigation cost algorithm from JPLs MER rovers is used to label training data.

Regression ML model accuracy is commonly measured using the Mean Squared Error (MSE) loss function [123], however during our analysis we have discovered that these residual errors were in some cases related to the true cost value itself. To visualise this relationship Confidence Band plots are used to describe the output quality of our models. The metrics used to describe

and measure the accuracy of these models is described in Section 3.3.2.

During this early feasibility investigation the execution time of our models was measured using the desktop platform used for development, two metrics could be measured reliably the execution time of the model and the size of the models weights. These metrics are clearly of limited use given our eventual target platform is a radiation hardened LEON processor, but they do indicate the relative speed of the models. Chapter 4 addresses these concerns by describing the deployment tool that was developed which produces LEON compatible C implementations. Building and executing these models on a LEON emulator produced the accurate performance metrics presented in later chapters.

3.3.1 ML Models Evaluated

Common image processing CNN models can be adapted to generate cost-maps since they process input matrices of scalars and estimate the continuous scalar navigability value. Our early work [22] used these regression CNNs for cost mapping and found them to be effective at producing accurate results, but required more processing power than the baseline algorithms they were intended to replace. It was realised that using a whole ML model to estimate the navigability of a single location would never be as efficient as the hand crafted baseline algorithms. After surveying a range of ML models it was decided to evaluate Encoder Decoders models [30]. These models can process input tensors into output tensors so they should be able to estimate an area of navigation cost elements in a single inference pass.

A range of CNN and Encoder-Decoder models of various sizes will be evaluated to measure the effect that a model size has on its accuracy. Each of the models listed in Table 3.3.1 are described in terms of their layers, filter sizes and activation functions, detailed descriptions of these models can be found in Appendix A. These models have been defined in such a way that their size can be automatically scaled. This is done by altering the number of convolution filters and sizes of dense layers for each model. This technique has been used so that the computational cost of models can me compared to the accuracy of estimates they produce.

Figure 3.3: High level description of the five CNN models evaluated in our AHS publication [22]. Detailed descriptions of these models can be found in Appendix A

.

CNN Regression Models

A standard CNN regression model processes a two or three dimensional input tensor into a scalar estimate Figure 3.3. This is done using an input stage comprising several convolution layers followed by one or more fully connected layers. The five CNN models presented in our AHS conference paper [22] have been included to show the progression of our work.

Encoder Decoder Models

Spatial Encoder Decoder models are used in applications such as segmentation [30] [10]. These models use transposed convolution layers (commonly known as deconvolution layers) to expand the spatial size of a layer in a manner complimentary to convolution layers. In our application transposed convolution layers are used to expand fully the connected layers into the larger final output matrix of navigation cost estimates.

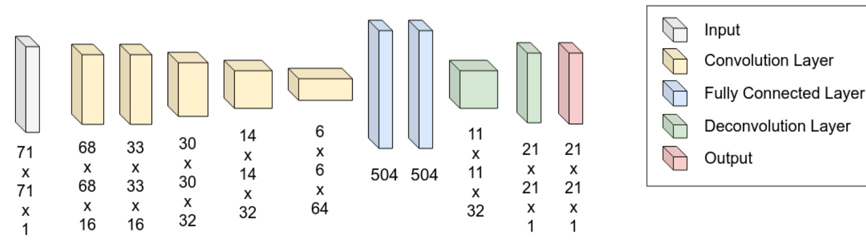


Figure 3.4: Topology of an encoder decoder model E used in this work. Here two transposed convolution layers with 5×5 filters and strides of 2 have been used to expand the output of the model.

.

Table of Evaluated Models

The following set of terrain assessment models have been evaluated, Table 3.1 summarises the structure of each model, number of weights and MAC instructions needed to perform an inference pass. Appendix A includes the complete description of each model.

Table 3.1: Set of eleven model topologies trained and analysed during our work.

Label	Topology	Cells Estimated	Weights (MB)	MACs (M)
Cnn-A	5× Conv 3× Dense	1	7.9	10.5
Cnn-B	4× Conv 3× Dense	1	16.9	12.5
Cnn-C	4× Conv 3× Dense	1	9.8	10.7
Cnn-D	3× Conv 3× Dense	1	39.7	16.3
Cnn-E	3× Conv 3× Dense	1	7.2	8.0
EncDec-A	5× Conv 2× Dense 1× Deconv	121	9.9	16.4
EncDec-B	5× Conv 2× Dense 2× Deconv	441	13.2	24.2
EncDec-C	5× Conv 2× Dense 3× Deconv	1681	16.9	43.1
EncDec-D	5× Conv 2× Dense 1× Deconv	121	3.1	14.6
EncDec-E	5× Conv 2× Dense 2× Deconv	441	5.7	22.3
EncDec-F	5× Conv 2× Dense 3× Deconv	1681	9.2	41.2
EncDec-X	5× Conv 2× Dense 3× Deconv	7921	23.6	26.0

The five CNN models studied during my earlier work vary the number of 3×3 convolution layers used in the input stage and the size of full connected layers in the output stage. It can be seen that the number of weights and MACs increases as the number of convolution layers in decreased, this is caused by the size of the output of the the final convolution layer increasing. The size of this layer in turn affects the size of the following fully connected layer.

Six Encoder Decoder models have been investigated, they all share a common input stage of five 3×3 convolution filters and two fully connected layers. The difference between them is in the number of transposed convolution layers in the output stage and the size of the filters used in these layers. Encoder Decoder models A, B, and C use 3×3 transposed convolutions while D, E, and F use 5×5. This was chosen so the effect of the expansion ratio of the output stage and the filter size could be studied. It should be noted that the Encoder Decoder models have larger input matrix sizes than the CNN models because the input must always be the same amount larger than the output due to the nature of the cost mapping problem as described in Section 3.11. This relationship between input and output sizes is the reason for the majority of the increase in weight and MAC count of these models.

3.3.2 Model Evalulation

The accuracy of regression models of the type investigated is commonly measured using the MSE loss function [123]. This metric is used internally by Huber loss function during training however different metrics are used to evaluate the accuracy of models once they have been trained. The requirement given to us for the cost-mapping process is a maximum error of 10% at three sigma, this threshold can be visualised using a 99.7% Confidence Interval (CI) where 99.7% of estimates are expected to fall within the 10% error requirement.

Cost-map estimate errors were discovered to be non-uniformly distributed as shown by the 2D histogram in Figure 3.5.a. It can be seen that the standard MSE metric or a scaling of it to 3 sigma will not capture the peak errors at different true cost-values. Visualisation using 2D histograms is useful but is distorted by the non-uniform density of true cost-values, shown on the x axis in Figure 3.5.a. For this reason Confidence Band (CB) plots have been used to visualise the detailed accuracy of trained models in this work as shown in Figure 3.5.b. These plots are immune to variations in true cost-value density and clearly show the worst case errors at the 3 sigma requirement. The results shown in Figure 3.5 are from an early poorly performing model which has been chosen to highlight non-uniform errors. The estimates can been seen to be more accurate over costlier terrain except for extremely hard or impassible terrain where the error increases.

While confidence band plots such as Figure 3.5.b are an insightful description of a models

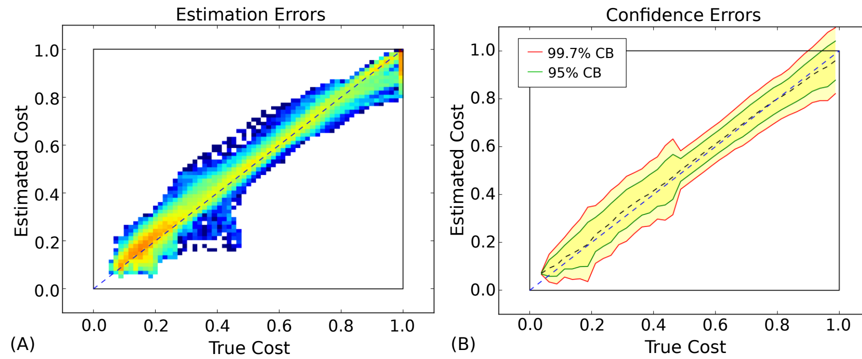


Figure 3.5: a) 2D histogram of ML estimates against true cost-values. b) Confidence bands of navigability estimates.

Table 3.2: Scalar metrics for example model

Worst 99.7% CI	**Mean 99.7% CI**
0.1703%	0.1211%

performance, scalar metrics are needed for the quantitative comparison of models. For this purpose two metrics have been used, the worst 99.7% confidence band error and the mean 99.7% confidence band error. These quality metrics for the early model described in Figure 3.5 are shown in Table 3.2, showing that this model does not meet the 10% requirement.

3.3.3 Datasets

In order to train and evaluate ML models for navigability estimation it is necessary to have datasets of elevation maps and their associated navigation cost maps. These are not common datasets and nothing suitable for this work is publicly available. This Section describes the sources of data that have been used in this work and the processing steps that were necessary before they could be used for training ML models.

Two proprietary datasets were made available by Airbus for us in this work; A sets of localised stereo images pairs from an Exomars breadboard rover traversing the Stevenage Mars Yard, and a high resolution DEM from the ERGO dataset collected by an ESA consortium. A further dataset was collected ourselves using a hybrid LIDAR camera sensor in also in the Stevenage Mars Yard.

Although all three of these datasets were collected using different sensors and have been processed to different levels, they all essentially describe the geometric shape of a planetary analogue terrain. This has meant we have had to generate our own navigation cost maps for these terrains. The published navigation cost algorithm using on JPLs MER rovers was used for this task as described in Section 2.2.2.

The decision to collect our own dataset was motivated by the need to publish our work, we were able to use the proprietary datasets for research but were unable publish this data alongside it. In the interests of repeatability we wanted to be able to publish a fully working system including code and data to disseminate our findings. Since this was not possible with the data initially made available to us we decided it was worthwhile collecting our own. This terrain dataset was collected using a hybrid camera and 3D LIDAR sensor and is described in Section 3.3.3.

The following sections describe each source dataset and the processing required to generate a usable DEM from it. Finally four different maps are described, two of which are regions of the larger ERGO dataset, these four maps were used for all the analyses presented in this chapter.

Stevenage Mars Yard

Figure 3.6: The Mars Yard test terrain at Airbus Stevenage.

Two DEMs have been produced of the Airbus Mars Yard in Stevenage, Figure 3.6, the first was generated using stereo image pairs while the second was produced using a hybrid LIDAR camera sensor. Early in our work, before we had access to a 3D LIDAR or the Mars Yard facility,

Airbus provided a set of navcam stereo image pairs taken from a breadboard rover traversing the room. Some time later we had built a working 3D LIDAR camera sensor, Figure 3.8, which we used to collect our own dataset at the Mars Yard in person.

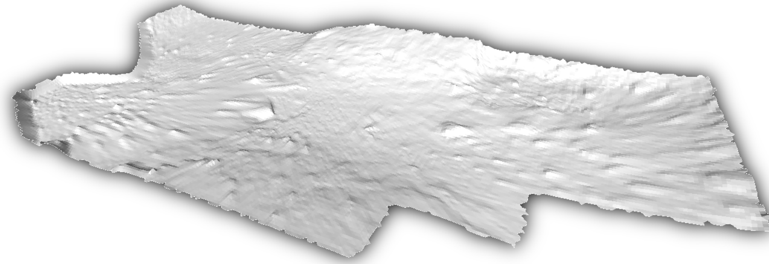


Figure 3.7: Low resolution DEM of the Mars Yard generated using photogrammetry. Note the lack of fine detail, especially around the edges of the map.

The first dataset was produced using photogrammetry to generate a triangulated mesh from input images, this mesh was then converted into the DEM shown in Figure 3.7. This was the first representative planetary terrain model which was used to train our DL traversability estimation models, [22]. However this terrain model suffered from a problem common to photogrammetric reconstructions, fine details were lacking resulting smoothed rocks and poor differentiation of rocks and sand. We were concerned that this smoothness would make this dataset easier to train than one with more detail. In order to test this hypothesis we decided to create our own higher quality reconstruction of the Mars Yard.

Our high resolution LIDAR dataset of the Mars Yard was produced using a scanning LIDAR with a 1 cm depth precision. It is far easier to produce highly detail reconstructions using this type of sensor, that doesn't suffer from the non-linear depth error of stereopsis. Our custom built LIDAR camera sensor comprised an 80 metre range Hikoyu scanning LIDAR and GoPro monocular camera both mounted on a precision tilt platform, Figure 3.8. This sensor can produce a variety of data products, for this dataset we collected colour 3D point clouds. Forty five of these were captured at a variety of locations and headings around the Mars Yard, which were aligned using Iterative Closest Point (ICP) and converting into a high resolution DEM. As can be seen in Figure 3.9 the resulting DEM contains far more fine details than the original stereopsis reconstruction. These two datasets have been used to compare the effect that detail

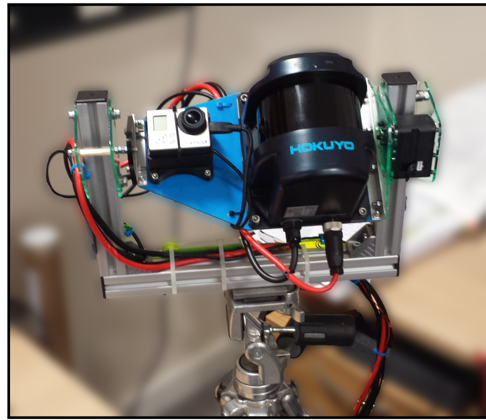


Figure 3.8: Custom made Lidar camera fusion sensor, used for detailed mapping of the Stevenage Mars Yard.

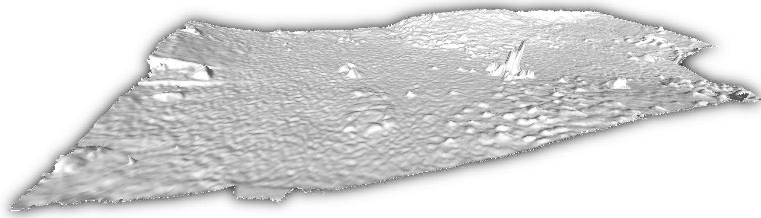


Figure 3.9: High resolution DEM of the Mars Yard generated from LIDAR data. Rocks and edges are much more clearly defined than the earlier map generated using photogrammetry.

level has on the difficulty of training a terrain assessment model on them.

Unfortunately Brian the mass simulator breadboard had been left in the middle of the test terrain, and we were unable to drive him out. So our map has a large and unusual shaped rock in the centre of it, luckily this doesn't negatively impact our terrain assessment task since it is treated as just another obstacle, albeit and unusual one.

ERGO Dataset

The European Robotic Goal-Oriented Autonomous Controller Field trials took place in the Moroccan Sahara desert during November to December 2018. The primary goal of this trial was a 1 km autonomous traverse conducted by the SherpaTT rover. During these field trials a high quality DEM of the site was generated from drone captured images, Figure 3.10, this

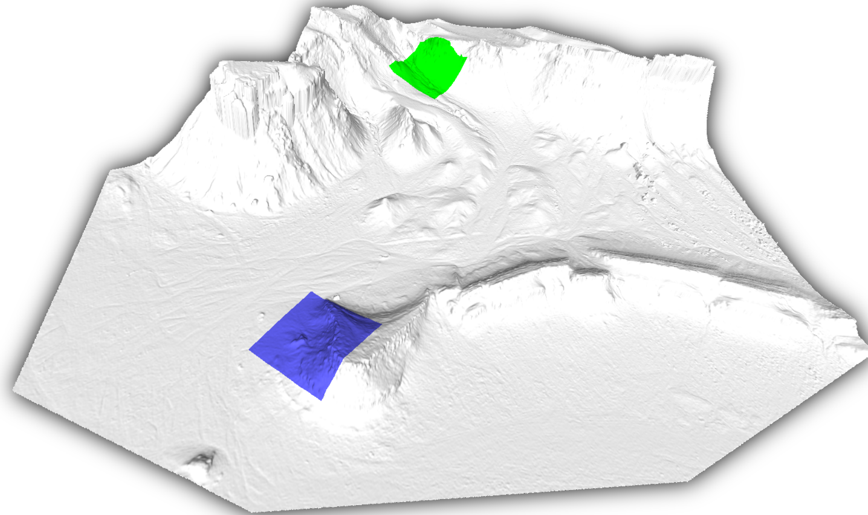


Figure 3.10: Terrain DEM from the ERGO field trial site, covering an area of approximately 300 x 300 metres. Two sub regions of the ERGO terrain map have been used for model training and evaluation. 'Slope Hill' region is shown in blue, 'River Bed' region is shown in green.

was compared with the map generated on-board Sherpa to evaluate the performance of its Simultaneous Localisation and Mapping (SLAM) system. This DEM is large, approximately 300 by 300 metres, and covers a terrain of rock and sand with virtually no vegetation.

We have used several regions of this full dataset to train and evaluate our models, although this map is significantly larger than the Mars Yard and has a wider variety of terrain it has less fine detail than the other two maps. Two sub-maps of the full ERGO DEM were extracted for use in our work, a sloping hill feature blue highlight in Figure 3.10 and a gully highlighted in Green. These both include a range of different types of obstacle and overall navigation difficultly. Detailed views of the 'Slope Hill' and 'River Bed' sub maps are shown in Figure 3.11.

3.3.4 Pre-processing and Data Augmentation

The navigation cost function used in the GESTALT rover GNC system as described in Section 2.2.2 has been used to generate the navigation costs maps used to train our models. The resulting input data and labels for all our models are 2D matrices of DEM and cost map values as shown in Figure 3.12. CNN regression models which estimate a single cost value are treated as a

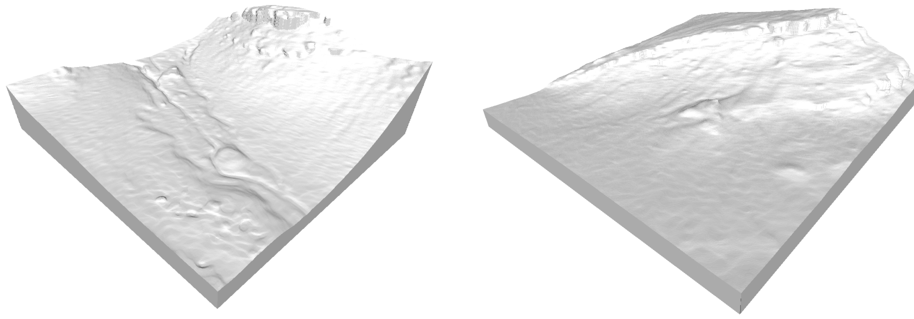


Figure 3.11: Ergo 'River Bed' terrain on the left, 'Slope Hill' terrain on the right.

special case where the costs label is a 1×1 matrix. The input matrix is always larger than the output matrix because these cost maps use obstacle expansion [45] based upon the size of the maximal rover footprint. The exomars rover maximal rover footprint is a circle 1 metre in diameter, when expanded to a square each cost map element can be effected by a 51×51 matrix of elevation values (representing a 2m x 2m square of the 4 cm resolution DEM, so the input matrix always has 50 more rows and columns than the output matrix.

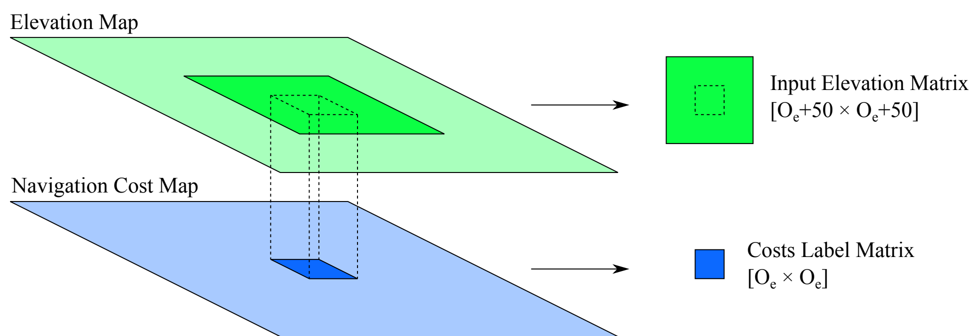


Figure 3.12: Extraction of a single training pair from a registered DEM and navigation cost map. Where O_e is the output edge length.

Before this input matrix of elevation values is passed to the ML model it's absolute elevation information is removed as described in Section 3.3.5. The mean of the input matrix is subtracted for each element, forcing it to have a mean elevation of zero. When model inference is being evaluated this is the only pre-processing step required, however during training additional steps are taken to artificially augment data.

Data Augmentation

The two stereo and lidar Mars Yard datasets contain 135k and 154k training sample respectively, while each sub-map of the ERGO dataset include 487k, this a significant amount and is enough to successfully train our navigability assessment models, however more training data will always improve the quality of trained models. Since the terrain assessment task is rotation agnostic and symmetric it is trivial to use reflection and cardinal rotations to increase the size of these datasets eight times.

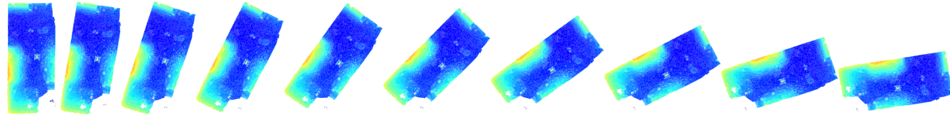


Figure 3.13: Rotation augmentation of the Mars Yard lidar dataset, with rotations from 9 to 81 degrees. Additional augmentation using reflections and cardinal rotations is performed online during training.

Table 3.3: Sizes of training datasets with cardinal rotation augmentation and arbitrary rotation augmentation.

Dataset	Original	Cardinal Rotations	Semi-Arbitrary Rotations
Mars Yard Stereo	135k	1,077k	10,185k
Mars Yard LIDAR	154k	1,235k	11,733k
ERGO River Bed	487k	3,898k	37,591k
ERGO Slope Hill	487k	3,898k	37,591k

These cardinal rotations were implemented in the dataset provider code directly, since they are simple and quick to apply to the data as it is being fed into the model. However the terrain assessment task is agnostic to any arbitrary rotation so we can augment our dataset more than just eight times. Due to the complexities of the non-uniform maps we are working with the simplest method to add semi-arbitrary rotations was to expand the original source DEMs with nine rotated copies of itself at angles from 9 to 81 degrees. This further multiplied the size of our training sets approximately ten times, resulting in the final sets that were used for model

training and evaluation shown in Table 3.3.

3.3.5 Absolute Elevation Invariance

DEMs represent the shape of terrain using a grid of elevation values, these values are defined with reference to a vertical datum. In the case of the two Mars Yard datasets this datum is the floor of the room it is housed in, while for the ERGO dataset it is mean sea level. This entirely arbitrary difference means that elevation values in the ERGO maps are approximately 800 metres higher than the other two maps. This difference serves to highlight that the absolute elevation of a map region should have no bearing on its traversability estimates.

This elevation invariance can be seen in the baseline GESTALT algorithm which produces its estimate based upon the overall slope and relative elevations of a DEM patch. This section investigates two methods by which absolute elevation invariance can be built into an ML model and compares their performance. The techniques of weight normalisation [122] and layer normalisation [9] were both adapted to enforce elevation invariance, and their effectiveness for this application evaluated.

Weight normalisation was adapted by applying mean only normalisation at the first layer such that each individual filter has a mean of zero. This forces filters to be invariant to the absolute values of their inputs. Removing any absolute elevation information in the first layer like this means it is impossible for it to have a bearing on the final estimate. This technique differs from weight normalisation as described by Salimans et. al. [122] because only the mean is normalised not the variance and each filter is normalised individually.

Complimentary to weight normalisation, layer normalisation can be applied to the input elevation matrix, Hinton et. al [9]. Again mean normalisation is performed, because the variance of the elevation values describes information relevant to navigability. Superficially these two techniques look symmetric however they are mathematically different and will be shown to perform differently.

The adapted weight and layer normalisation techniques have been evaluated alongside a base case where no elevation invariance is enforced. These three input configurations are evaluated using the four maps: Mars Yard stereo, Mars Yard LIDAR, ERGO slope hill, and ERGO river

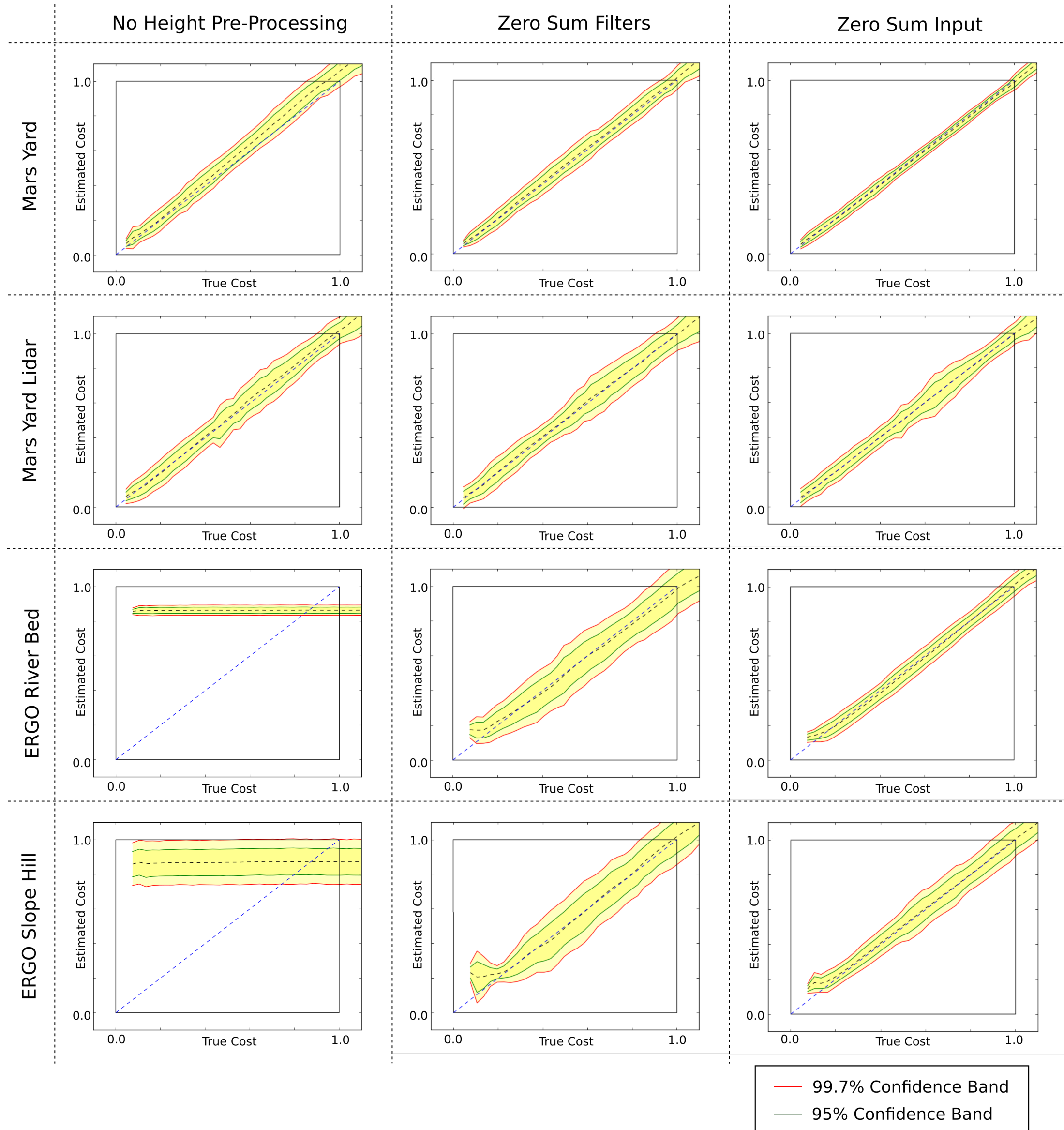


Figure 3.14: Navigation estimate confidence bands for the encoder-decoder-A model trained on all four test terrains, with either no elevation invariance, weight normalisation, or layer normalisation. It can clearly be seen that layer normalisation produces consistently better models, although weight normalisation is also an improvement over the baseline. The models trained on maps from the ERGO dataset failed to converge at all without one of elevation invariance techniques being applied.

bed. Figure 3.14 shows the confidence bands for these twelve experiments, their mean and worst errors at 3 sigma are shown in Figure 3.15.

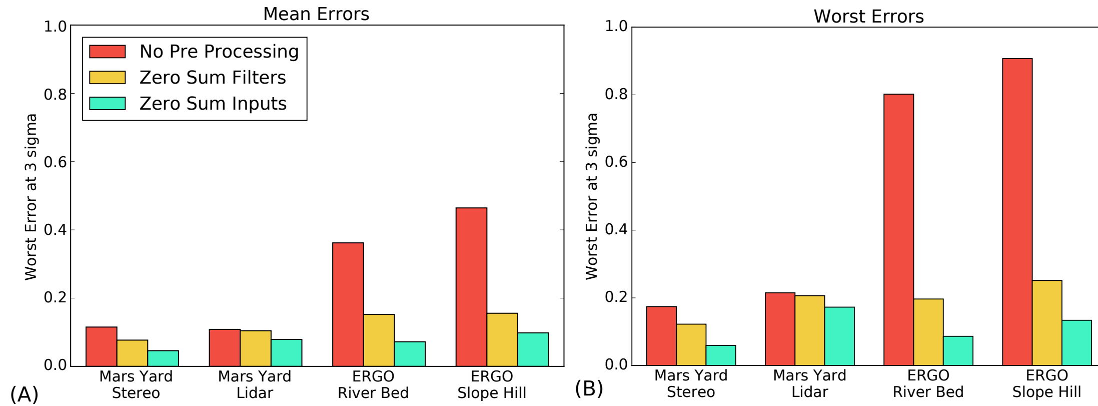


Figure 3.15: Comparison of elevation invariance pre-processing techniques on all four training maps. A, shows the mean error at 3 sigma. B, shows the worst error at 3 sigma.

When training Encoder Decoder model A on all four of the test terrains, weight normalisation offers some improvement over the baseline, however layer normalisation provides the greatest improvement in all cases. This is a minor engineering contribution, as far as the author is aware these techniques have not been used before, but they represent a small incremental development upon existing techniques.

A possible explanation for the difference in performance between the two techniques is that the layer normalisation preserves the relative elevation information across the whole of the input matrix. weight normalisation meanwhile only preserves the relative elevation of the input within a single filter kernel, 3×3 or 5×5 for the models tested. The additional information that is preserved by layer normalisation could explain why it produces most accurate trained models.

3.3.6 Model Training

Cost-mapping models described in this chapter were trained for 10k steps with a batch size of 100. The Huber loss function was used with a delta parameter of 0.45 [160]. In this regard the training configuration used is quite standard, however there are some important differences to which are described here.

The division of training data into train, test, and evaluation sets needed special attention because

of the nature of this task. A subtle improvement was found if the dynamic range of the cost value was allowed to extend past 1.0. If this cost-map generation technique is used as part of a practical GNC system then an adapted loss function should be used due to the consequences different types of estimation error.

Division of Train, Test, and Evaluation

When training a model it is critical to evaluate its performance on data which is representative but not the same data it was trained on. If a model is evaluated on the same data it was trained on then over-fitting can be mistaken for increased accuracy. During supervised learning it is common to split training data into three sets, Training data used to train the model, Test data used to evaluate the model in between training steps, and Evaluation data used to ultimately measure its trained accuracy.

In most cases these sets can be created by randomly splitting the data because each training case in independent. However in our case because each element of the cost-map depends on many adjacent elements of the input cost-map they are not independent. If training data was split randomly then during training the model would be exposed to all the input data of the test and evaluation sets, means over fitting could not be detected.

In our cost-mapping experiment the Test, Train, and Evaluation sets were produced by selecting contiguous regions of the training cost-map and discarding elements along the boundaries of these sets. The 'width' of the discarded region along this boundary was 25 elements, because $(ReceptiveFieldWidth - 1)/2 = 25$. These boundaries although wasteful of training data mean that the receptive fields of samples from the three different sets will never overlap.

Extended Dynamic Range

Two observations lay behind the idea to use extended dynamic range training data. The first was that estimation errors in earlier experiments were often greater at the limit of the true cost values, near 1.0. The second was that this navigation cost limit is actually artificial. It is imposed as the safe limit of the rovers capabilities, it has no meaning in terms of the terrain itself. For example if your rover has a tipping angle of 25 degrees so a safe limit of 20 degrees has been chosen to

have a cost value of 1.0. Here two slopes of 20 and 40 degrees would both have a cost of 1.0 even though the 40 degree slope is clearly more difficult to traverse.

It was found in our work that if training cost-maps were generated with values extending beyond 1.0 and this cap was instead applied after the ML model that accuracy increased slightly. The increase in accuracy was most pronounced around the 1.0 limit which as described in Figure 3.16 is the most critical region for errors.

Training for Flight

The goal of many ML models is to match an existing algorithm or dataset as closely as possible, image classification or segmentation for example. It may seem non-intuitive at first but the goal of a cost-map generation algorithm is not to estimate the navigation cost of terrain as accurately as possible. It is in fact to to help the rover reach its destination quickly and safely as part of the whole GNC system. This is a subtle but important distinction because it means we have to understand estimation errors not as an abstract scalar metric but in terms of their consequences on rover behaviour.

Four distinct error regions can be identified based upon the sign of the error and the true navigation cost of terrain, Figure 3.16. Exaggeration errors are defined as erroneously thinking terrain is harder to traverse than it actually is, but not so hard that it will never be traversed. The effect on the path planner in this region is to force it away from optimal route in some cases, there is no danger to the rover and no gross effect on planned routes. The same effect is caused by the Understatement error region, the path can be forced through harder terrain than expected but not to the point that the rover is in danger.

The final two error regions have more serious consequences. The False Wall region is defined by the cost-mapping algorithm erroneously marking safe terrain as impassible. This can have the effect that a region of safe terrain is excluded from any planned paths potentially forcing routes to be far longer or even fail to be found at all. This can slow the rover down significantly or force ground control to intervene.

Danger Zone errors are classified as where the cost-map shows terrain is safe where in reality it is hazardous to the rover. Routes could be planned and executed which put the rover at risk

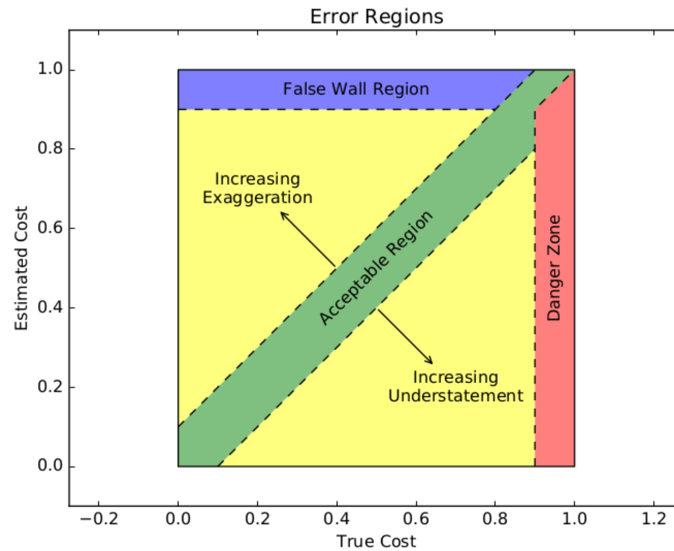


Figure 3.16: Map of error regions divided by their effects.

of becoming trapped or damaged. While there are additional safety systems which can abort a traverse if it appears to be unsafe, they are not foolproof and even a small risk of a mission ending event must be avoided is possible.

The MSE loss function was used in this work to theoretically demonstrate that ML models are capable of generating cost-maps. However the MSE metric does not capture the varied consequences of estimation errors, if this technique is used in a complete GNC system a utility cost function would be more appropriate. This could potentially be implemented by artificially increasing the magnitude of errors in the danger zone, or artificially increasing the truth-cost values around 1.0 to 'push' estimates away from the danger zone. Further research into this training detail would be a valuable addition to the field.

3.3.7 Initial Inference Timing

The industrial case has a requirement that the cost-mapping process for a complete local map must complete within 20 seconds on the LEON3 processor as described in Section 3.2. At this stage in our work however there were no tools available to deploy ML models to this target [22]. It would have been possible to hand code these models, however the time required to implement and verify these models was considered prohibitive. Since it was not possible to implement

these models on the LEON3 processor no performance measurements could be taken.

A crude initial timing of inference execution was performed on our desktop development platform, an Intel 8 core i7 PC with an Nvidia GTX 1070 GPU, these results are shown in Section 3.5.2. Each model was built with a batch size of one and inference was performed a thousand times, the median of these runs was then taken. These performance results do not tell us anything about the execution time on the target radiation hardened processors but they do provide an indication of the relative execution time of each model.

The amount of memory required to store the weights of each model is easy to measure from the models flow-graph, while the amount of memory required to perform inference with each model is harder to determine. No mechanism exists within the TF library to compute the RAM required to perform inference. Given the large overhead of the library itself it was not possible to measure this using external tools. The TFL library was considered to measure RAM use however this is intended for mobile and tablet targets so it still has significant overheads of over 100 MB as described in Section 3.4.1.

The tools available to us are a great limitation at this point, the only measurement we can make accurately is the trivial size of model weights. Neither the execution time or RAM can be measured for our target platform.

3.4 Inference Feasablilty on LEON Processors

Evaluating the accuracy of ML models used for cost-map generation is only a part of our task. Inference using these models must to executed on the LEON3 processor, and this deployment has to be performed using a software process that is suitable to the space industry. This is where we step outside of the usual realm of ML and investigate the deployment of models on a platform which to the authors knowledge has never been done before. Our motivation for this task is twofold, to evaluate if the models presented in Section 3.3.1 are feasible to execute on LEON3 processors. Secondly to study the ML deployment process necessary for ML to be used within the software development and V&V processes of the space industry. An overview of the flight software development process for high-value missions is described in Section 2.5.1.

Model performance was measured on the desktop platform that was used for training as described

in Section 3.3.7. This is a crude relative measurement, and not applicable to the LEON3 platform we are targeting. Section 3.4.1 describes our attempts to implement these models on the LEON3 using existing tool which were ultimately unsuccessful. These limitations were the reason why these crude initial timing measurement were taken.

Section 3.4.2 reviews the latest Edge ML deployment tools against the requirements discussed in literature review Section 2.5.1, it has to be noted that these tools were not available at the start of this research. If they had existed then they could have been used to measure model performance on the LEON3, but would not be suitable for use in the flight software development process.

3.4.1 LEON Deployment Using Existing Tools

In early 2018 when our investigation into ML cost-mapping models started there were a range of software tools aimed at deploying ML models onto platforms smaller than desktop computers. The majority of these are aimed at phone & tablet scale targets, although the Tensorflow XLA AOT compiler is capable of targeting more constrained platforms.

C++ ML Deployment Frameworks

Two frameworks which deploy ML models to C++ code were evaluated, the TFL tool and the MatLab statstics and machine learning toolbox. Both of these tools are able to produce C++ projects implementing trained ML models. However they are aimed at desktop or mobile platforms, and as such they have dependencies upon large runtime libraries (greater than 100MB in the case of TFL) making them impossible to deploy to the LEON3 development boards we have access too. The Pender GR-XC3S LEON3 development board has 8 MB of flash PROM and 64 MB of Synchronous Dynamic Random Access Memory (SDRAM).

The size of run-time libraries required by both TFL and the MatLab statstics and machine learning toolbox ruled out both these options. Even if this had not been the case, the challenge of building the support libraries for the bare-bones Sparc V8 LEON3 platform would have been a non-trivial.

Tensorflow XLA

TensorFlow XLA is an LLVM compiler tool which generates object binaries directly from a trained TF model. The resulting compiled static-libraries can be linked into larger projects with minimal library over heads. This process is suitable for deploying our models on to the LEON platform, because use of the versatile LLVM compiler architecture [90] means it can produce Sparc V8 binary code, and it has minimal library overheads.

The Sparc V8 architecture of the LEON family of processors is not explicitly supported by the XLA tool, after some initially promising results the authors failed to successfully integrate it with the Gaisler LLVM compiler [53]. The Gaisler Clang compiler is built on LLVM version 4.0 while the TF XLA compiler generates IR using features only present in versions 5.0 and above. By taking the LLVM IR produced by the XLA compiler then building it using the Gaisler clang compiler simple fully connected networks were successfully deployed to the LEON3. However LLVM version incompatibilities prevented larger dense or convolutional models from being deployed using this tool.

The XLA compiler process cannot be integrated into the flight software development process, it produces either IR or compiled objects. Since it does not generate human readable code its output cannot be re-implemented to ECSS standards. Given the tool itself has only recently (2019) been moved out of the experimental part of TF, it is far too immature to be considered a reliable compiler in of itself.

Hand Coding

Alongside the automated deployment tools described in Sections 3.4.1 & 3.4.1 there is always the option of manually implementing any given model without using any support libraries. In practice this would involve writing and testing functions for each type of layer used by the model, then writing the core model flow graph itself. This approach would be practical for a one-off small model but quickly becomes untenable and error-prone for larger models. It was concluded that the effort needed to manually implement all the models described in Section 3.3.1 was greater than the effort to build a tool to do the same job.

3.4.2 Tools Released During this Work

During the course of this research two open source Edge ML tools were released which are capable of deploying models to the LEON3. They came about too late to be used for the timing measurements of our models, however it is worthwhile to review their capabilities in terms of the flight software development process described in Section 2.5.1. As is described below both of these tools are aimed at the rapidly growing field of Edge ML IoT applications, so are unsurprisingly not tailored to the rigorous reliability requirements of the space sector.

TFL micro

Spearheaded by Pete Warden the TFLμ tool was first released in January 2019 as an experimental module of Tensorflow V1.13.0 [116]. This tool enables TFL models to be deployed onto micro-controller scale targets, and is tightly integrated into the wider TF workflow. An interpreter paradigm is used, with models being encoded into the binary as data then loaded and interpreted using generic execution code. Use of dynamic memory allocation is avoided by using a single area of stack memory known as the tensor arena to store working data, which it manages itself. It requires a C++ 11 compiler and has dependencies upon the wider TFL code base.

TFLμ is a powerful tool with a growing user base, many of whom work on IoT projects, and targets CPUs smaller than the LEON3 processors we are work with. There are two fundamental obstacles to its use within the flight software development processes described in Section 2.5.1 though. Firstly is the C++ 11 requirement, which would not be trivial to remove, and secondly it is a significant code base itself while also depending on the larger TFL code base. Because it uses an interpreter paradigm a significant amount of code would need to be re-implemented to flight standards regardless of the complexity of ML that was being used.

μTensor

Lead by Neil Tan μTensor was first released in July 2018, and is primarily aimed at the Advanced RISC Machines (ARM) M-Bed range of micro-controllers, although the code it generates can be deployed on a wider range of targets. Unlike TFLμ μTensor uses a code generation paradigm,

producing C++ model implementations which depend on a small runtime library. μTensor is also designed to work with the TF framework, using TFL flatbuffer models as input.

μTensor is an ML deployment tool which generates highly efficient model code making use of ARMs optimised CMSIS library of layer implementations [88]. The code generation approach means that models depend on a smaller amount of code than TFLμ. However it still requires a C++ compiler and makes use of dynamic memory allocation, for these reasons it is not suitable for the flight software development process.

3.5 Results

The most important finding from this investigation is that standard ML models are capable of estimating terrain navigability from the same DEM inputs as traditional algorithms. Figure 3.17 shows the histogram and confidence bands of estimates vs truth for model 'EncDec-D' trained for 5k steps using the Mars Yard Stereo dataset. The model has converged well to the training data with a small and relatively uniform error. Figure 3.18 shows the baseline cost-map used for training and the estimated cost-map produced using the trained model.

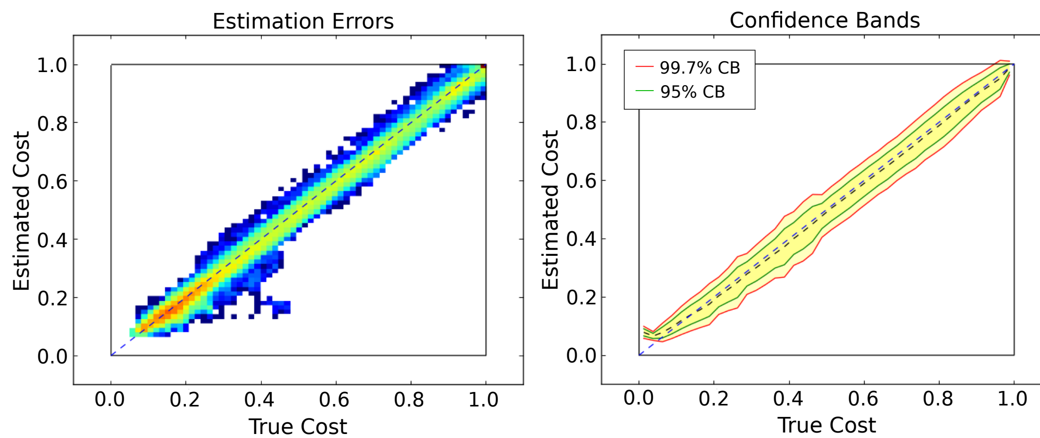


Figure 3.17: 2D Histogram and confidence bands of estimates against training values for the Encoder Decoder topology D trained for 5000 steps.

This model produces an 11×11 matrix of cost values on each inference pass, the boundaries of these can just be made out on the estimated cost map but the smoothness of the expected cost map reduces the significance of this. One of the penalties of estimating multiple cost values

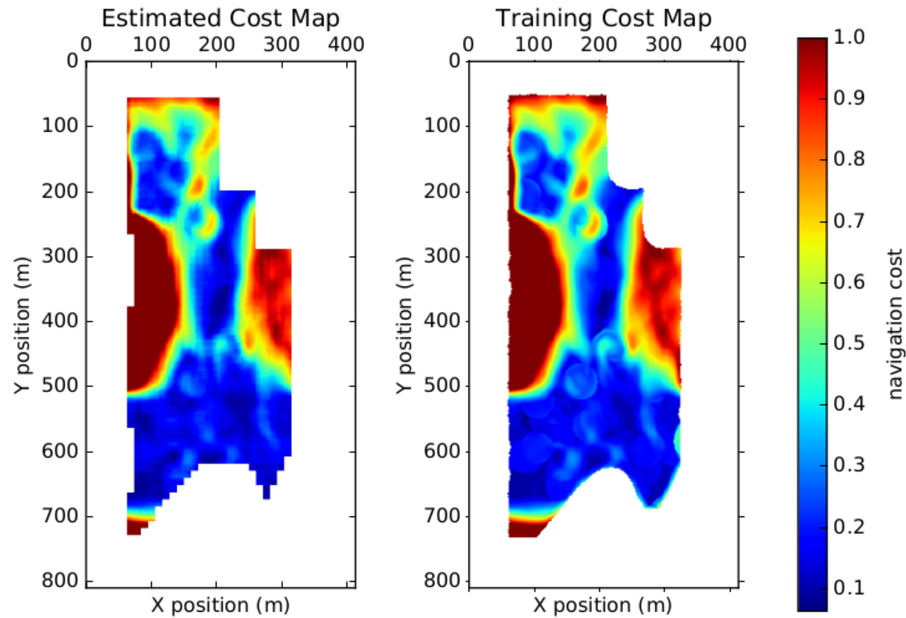


Figure 3.18: Cost map estimated using a trained Encoder Decoder model on the left in comparison with the ground truth cost map on the right.

at the same time is the requirement for a larger dense matrix of DEM values for the model to function. This is why the ML estimated cost map covers a slightly smaller area than the original training cost map, sparse DEM inputs around the border of this irregular map have been ignored.

The impact of this limitation increases as the output size and therefore input size of models increases. Encoder Decoder models C and F both produce 41×41 matrices of cost cells, which require a dense input matrix of 91×91 DEM cells. In practice the DEMs produced by rovers include unknown areas due to occlusions by rocks and other obstacles, therefore the current requirement for dense input matrices is problematic. Feeding sparse input data into an ML model is not common, however some work has been done in this area. Liu et al. [93] proposed a 'Partial Convolution' approach to the image in-painting problem which requires sparse input images to be processed. If such a technique could be adapted for these cost-mapping models, enabling them to operate on sparse inputs then these models would become more practical in a real world system.

3.5.1 Effects of Model Scale

Encoder Decoder model D has been shown to converge well to the training data in Figure 3.17, our requirement however is to meet a 10% error threshold at 3 sigma. A range of model types and sizes have been evaluated so that our later work measuring execution time is able to relate changes in accuracy with computational cost. All of the models described in Section 3.3.1 were trained for 5k steps with a batch size of 100, and their error measured using the Mars Yard Stereo dataset. This was repeated for different scales of each model between 5% and 100%.

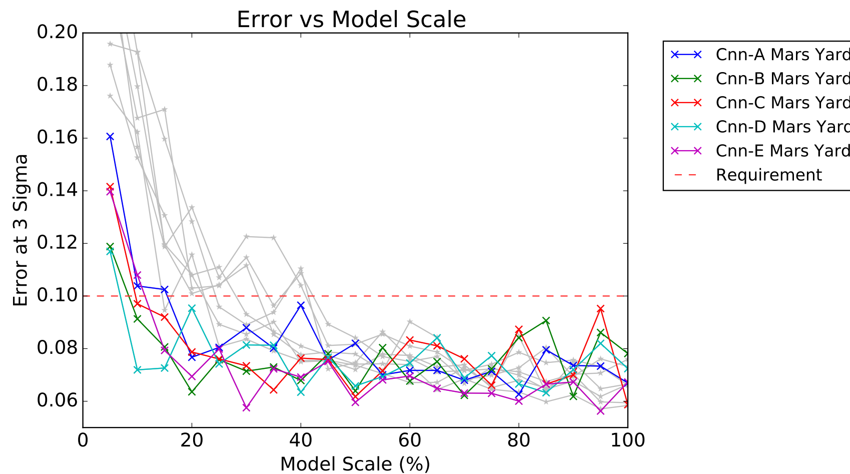


Figure 3.19: 3 sigma error results for all CNN models and scales. Encoder Decoder results are shown in grey for comparison.

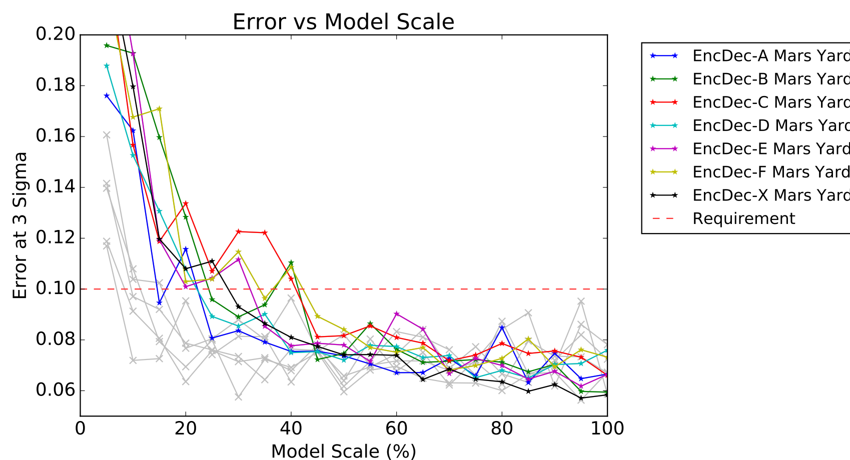


Figure 3.20: 3 sigma error results for all Encoder Decoder models and scales. CNN results are shown in grey for comparison

The 3 sigma error results for the five CNN models proposed are shown in Figure 3.19. Interestingly the errors in the cost-map values produced is largely consistent between the models themselves, this implies that the additional convolution layers present in CNN models A, B, and C are not necessary for good convergence. Scaling the model however does have an effect of the errors in their estimates. Model errors are relatively uniform between 100% and 20% of their full defined size, however below 20% a cliff edge can be seen where errors start growing rapidly.

These results indicate that it is possible to use the smaller scales of the smallest models proposed and still meet the cost-mapping accuracy requirements given. The execution time of these models on the LEON3 processor is not currently known, however this finding allows a wide range of effective ML models to be considered.

Error results for the seven proposed Encoder Decoder models are shown in Figure 3.20. These models are not as tolerant to down-scaling as the CNN models but are still able meet the requirement between 30% and 45% of their full size.

The key take away from these results is that existing ML techniques can be applied to the cost-mapping task and produce estimates that are accurate enough to be used by the downstream GNC processes on-board a planetary rover. Now that half of the question has been answered the second half of making these models work on space qualified CPUs can be tackled.

3.5.2 Costmapping Performance Results

Results shown in Figures 3.19 & 3.20 at first glance would imply that all thirteen proposed models are quite similar to each other. Our initial GPU timing results presented in this section show that there is in fact a significant difference between the performances of these models. With reference to the requirements defined in Section 3.2 the speed of these models is measured in DEM cells per second. This metric normalises these measurements against the changes in the number of cells estimated in a single inference pass.

Speed results of the thirteen proposed models shown in Figure 3.21 can be seen to have bunched together into five distinct groups. The cause of these groupings can be understood when they are compared to the number of cells each model estimates in a single inference operation, Table 3.1. Models in each of the five groups all produce the same number of estimates at a time, indicating

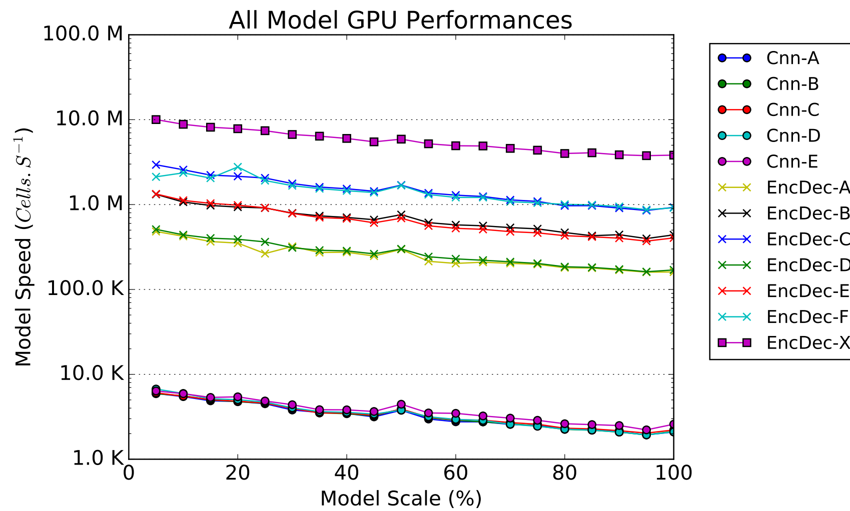


Figure 3.21: GPU performance measurements of all models collected on an Nvidia GTX-1070.

that this factor is the strongest contributor to the performance of these models. Therefore even though CNN models can be scaled down further than the Encoder Decoder models, this is insignificant when compared to their faster processing speed. The slowest of the Encoder Decoders is more than an order of magnitude faster than the fastest CNN model, even though this Encoder Decoder has lower estimation errors.

As mentioned before these performance measurements taken using a desktop GPU have little relevance to the LEON3 processor required by our industrial use-case. But it is worth noting that the highest performing Encode Decoder model X operates at between 5 and 10 million cells per second, more than three orders of magnitude higher than our requirement of 2405. Qualitatively this is in the right ball-park so that when it is implemented on the more constrained LEON CPU this requirement will still be met.

Such qualitative statements are of little value however, and without a deployment method or actual performance measurements on the LEON3 there is still uncertainty if this ML cost-mapping technique is feasible at all.

3.5.3 Comparison of Model Accuracies on Different Terrains

Section 3.3.3 discussed concerns that the level of fine details and range of terrain feature types contained in a DEM would have an effect on the ability of ML models to estimate cost-maps.

This experiment seeks to answer this question by training a set of identical models on the four terrain DEMs that are described in Section 3.3.3. These models were: Cnn-A, the largest of the CNN models EncDec-A, the smallest of the Encoder Decoder models studied, and the EncDec-D model a medium sized Encoder Decoder model. The results of this experiment are shown in Figure 3.22 where the error produced by each model has been averaged over all its scales.

Figure 3.22 shows there is a reasonably consistent relationship between estimation error and the terrain dataset used, across all three models investigated. The Mars Yard Stereo dataset was the estimated with the lowest errors, potentially because this map is smaller than the ERGO maps and lower in detail than the Mars Yard Lidar map. The size of this map means it has a smaller variety of feature types than the ERGO maps which may contribute to this difference. The ERGO maps of the River Bed and Slope Hill have the next higher estimation errors. These maps contain less fine detail that the Mars Yard Stereo map but are approximately three times larger, and contain a wider variety of feature types. Two of the three models tested produced their highest errors on the Mars Yard Lidar data, this map has significantly more fine details than the other three, indicating as suspected that maps with finer details are more challenging for an ML model to process into a cost-map.

Figure 3.22: 3 sigma error averaged across all model scales, shown for each training dataset and three model topologies.

These results imply two factors will be critical in the implementation of an ML cost mapping system on board any planetary rover. Firstly the model will need to be trained using DEM models generated using the same sensor configuration and processing pipeline as the final

rover. This will improve the performance of the resulting model and more importantly mean the measured accuracy of the model will be more representative. The second implication is that the nature of the terrain itself has an effect on the accuracy of the model. Any model developed for flight will need a significantly larger training set than any used during this work, which also contains a wider range of terrain types. Even with a large and varied training set there will be a risk of encountering unexpected Martian areography resulting in unexpected behaviour. This risk will need to be closely scrutinised before this system could be deployed on a flight mission.

3.6 Summary

Two standard ML models, CNNs and Encoder-Decoders, have been shown to be capable of generating navigation cost-maps. The accuracy of the maps produced when compared with the baseline algorithm can be controlled by varying the size of models used. Two adapted normalisation techniques have been shown to marginally improve the training of these models. Overall it has been shown that it is possible to generate cost-mapping data using ML within the 10% error at three sigma defined in our requirements.

A crude execution time measurement has been used to determine the relative speeds of these models. After repeated attempts to deploy models to the LEON3 processor using existing tools, it was determined that none of these tools were suitable. TFL and MatLab ML toolbox were large to be deployed on this processor, while Tensorflow XLA produced implementations at a realistic scale tool-chain incompatibilities prevented it from working. Due to these challenges none of the cost-mapping models have been executed on a LEON3 processor at this point, so their performance is unknown.

In addition to the practical barriers to using these deployment tools with the LEON3 platform it has been shown that they are incompatible with the software development and V&V process used for flight software in the space industry. Therefore even if it had been possible to use them for performance evaluation, new tools would still have been required before these ML solutions could be flown.

Progress has been made against the four opportunities identified at the end of Chapter 2 as shown in Table 3.4. The question of weather ML models are capable of producing cost-maps of

the required accuracy has been addressed. However the questions of performance and method of integration into the flight software development process remain.

3.6.1 Future Work

Several specific opportunities to improve the ML terrain estimators presented in this chapter have been identified. The models we evaluated can only operate on dense DEMs because there is no mechanism for defining unknown map cells in the input. Sparse DEM are expected in a practical implementation, especially if larger map areas are to be processed at once. Such processing was identified as the most computationally efficient implementation of this approach. The author feels this would be the most valuable next step in the ML aspect of this work.

Only one baseline terrain navigability algorithm has been used for training the models presented, this is a limitation of our work. The contemporary equivalents to the GESTALT cost-mapping algorithm are proprietary, neither of the algorithms used by Curiosity or Exomars have been publish in any detail. A valuable future step would be getting access to more advance traversability algorithms or enough output data and training these ML models on them. Some of these algorithms use different input information, for example the Exomars geometric maps include the minimum, mean, and maximum elevations within a cell as opposed to a single elevation. Additional non-geometric information has also been proposed, visual texture for example [67]. ML would be expected to be capable of integrating these additional sources of information with ease, but a thorough investigation is warranted.

The greatest weakness in our investigation so far is the lack of execution time measurements on the target LEON3 processor. These radiation hardened CPUs are much less capable than their terrestrial equivalents, and development tools are also more limited. The absence of performance results on these targets casts a great deal of uncertainty on the feasibility of this approach. The lack of an ML deployment system which is suitable for the flight software development process is also a hindrance to the final roadmap we wish to present enabling the adoption of this technology on future missions. It is for these reasons that the next step of our work will address the challenge of automatically deploying ML models to LEON3 processors and within the requirements of the flight software development process.

Table 3.4: Opportunities at the end of the cost mapping investigation.

Challenge	Solution
Is it possible to generate planetary rover cost-maps using ML?	• Encoder decoder models trained using supervised learning have been shown to be effective.
Are ML cost-mapping models feasible to use on radiation hardened LEON3 computers?	• Promising indications based upon GPU performance, but no accurate measurements have been made on the LEON3 processor.
Does a deployment process exist to implement this model within the flight software development process?	• Existing automated tools reviewed, none found to be suitable, hand coded implementation only option.

New Research Opportunities Identified in this Chapter

- A tool which can deploy ML models to the LEON family of processors and work within the requirements of the flight software development process would aid the adoption of ML solutions in the space industry.
- Study ML cost-mapping models capable of processing sparse input data into cost-maps.
- Train ML models on a range of map types and traversability algorithms to find how accuracy and model size is affected.

Chapter 4

TFMin Tool

4.1 Introduction

Progress of ML research like other technical disciplines is defined by the co-development of theoretical understanding and the tools necessary to gain that understanding. Chapters 2 & 3 identified a gap in the capabilities of ML deployment tools. These tools are unable to deploy models to the current generation of radiation hardened processors, or within the software development process used on high value space missions. This section will describe the development of the TFMin tool indended to fill this gap and the research tasks it facilitated. A detailed description of the tool which was ultimately developed is presented along with use cases of the ML deployment research conducted.

Deployment of ML models onto low power embedded computers and micro-controllers is a recent development in the time-scale of this work (2016 - 2020) with the first open source tools supporting this development being released in mid 2018 [116][5]. Our research started before these tools were made public so there was no alternative to developing our own tools. The resulting TFMin tool has now been released open-source to complement the tools provided by ARM and Google. These three tools each provide different sets of features supporting different areas of research and engineering..

The following sections detail the requirements and architecture of the TFMin tool, its specific features and the research drivers behind them. References are made to particular research

insights enabled by the features of this tool. Finally the computational requirements of the cost-mapping models proposed in Chapter 3 are analysed by deploying these models to an emulated LEON3 processor. These results are compared to our initial relative performance indications from Chapter 3, and the impact of these numbers on the practicality of on-board ML cost-mapping discussed.

4.2 Motivation

Interest is growing in the use of neural networks and deep-learning for on-board processing tasks in the space industry [27]. However development has lagged behind terrestrial applications for several reasons: Space qualified computers have significantly less processing power than their terrestrial counterparts. Reliability requirements are more stringent than the majority of deep-learning applications, resulting in strict V&V processes. These V&V requirements slow the adoption of new development tools and frameworks while they are evaluated. Finally the long requirements, design and qualification cycles of most space missions mean that the latest technology at the outset, will be obsolete by the time the mission flies.

We present an open source tool chain that automatically deploys trained inference models from the TF framework to a wide range of hardware targets including the LEON family of radiation hardened processors. The ANSI C code which is generated can be deployed to the widest possible range of hardware targets, including DSPs, micro-controllers and CPUs. Critically the code generation approach allows existing space industry V&V practices to be applied to the prototype code produced, avoiding the need to strictly verify the TFMin tool itself. This has already allowed this tool to be used in low TRL ML studies in the space sector.

4.3 Architecture

Fundamentally TFMin is a python library which provides the functionality to import and manipulate ML models then generate the source code of an ANSI C implementation of inference. The deployment process is comprised of three key steps shown in Figure 4.1. First models are imported either from a Tensorflow flow graph or TFlite flatbuffer and converted to $tf_min.Graph$

object containing a framework agnostic internal representation, Section 4.3.2. This representation is generic so that importers from other ML frameworks could be added, however this was not necessary during our work so is left as a task for the open-source community. The graph translation pipeline is a sequence of translators used to optimise the model and prepare it for deployment. These translators are functions which take an input graph and set of parameters, returning a modified graph. They can be as simple as removing drop-out operations automatically or as complex as quantisation. This step is described in more detail in Section 4.3.3.

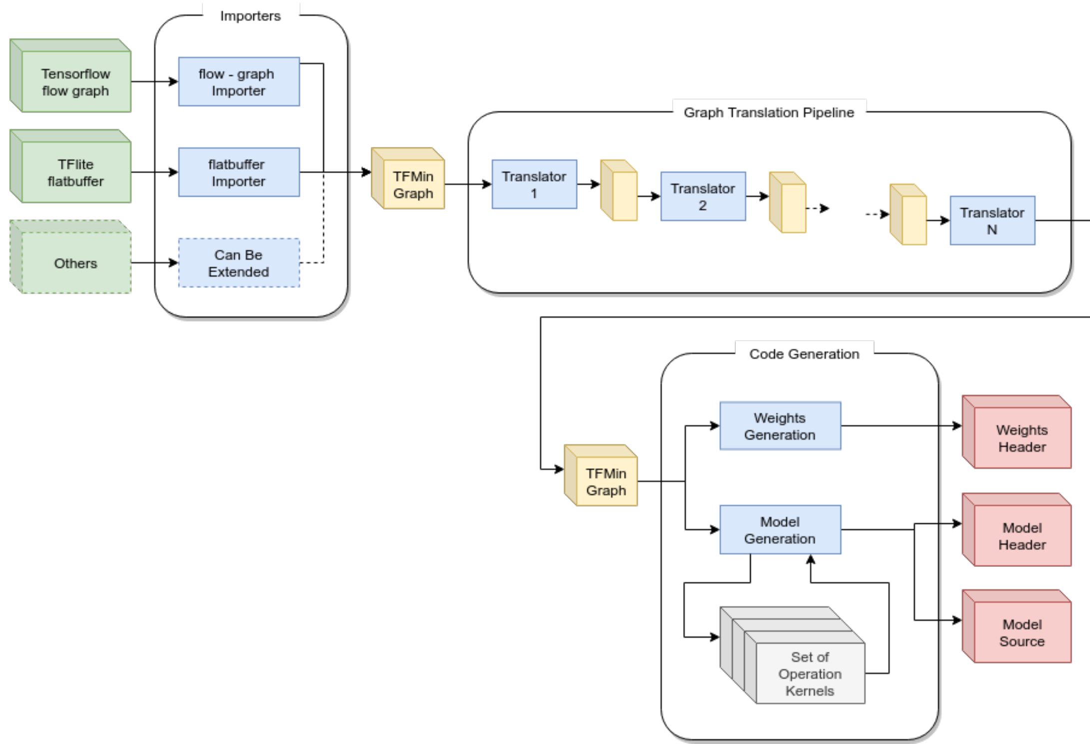


Figure 4.1: Data flow of ML model deployment using TFMin, showing the three top level processes performed.

The final step generates C source code using a library of operation kernels. Each of these kernel objects generates the C implementation of a class of operations. Intermediate tensors are stored at pre-allocated locations a single contiguous memory region, the tensor arena, allowing models to be deployed on systems that do not support dynamic memory allocation.

In addition to the standard deployment process described above, models can be deployed alongside introspection code to analyse the generated implementation in more detail. These analyses include per operation timing information and detailed memory access pattern visualisation and

are described in Section 4.4.

4.3.1 Design Rquirements

From a functional perspective TFMin is primarily a tool to convert a high level representation of a ML model into a pure C implementation of that model. However alongside this seemingly simple task it provides a wide range of configuration options, choices of algorithms to use at each stage, validation checks, and introspection tools. An open extensible framework enable developers to create entirely new graph-translators 4.3.3 and op-kernels 4.3.6.

To provide a consistent and easy to use framework for researchers to design ML experiments, a set of design philosophies was chosen to guide the development of TFMin. Several of these are simply good coding practice but they are described here and justified in terms of their effect on the project. These philosophies were combined with the requirements set out in Section 2.5.1, to inform the analysis and design of this tool.

1. Simple task simple interface, always use as few API calls as as possible and keep them simple. When writing the code for an experiment you want most of your code to describe experiment, not many lines of boilerplate set-up and tear-down.

2. Configuration settings are always exposed and controllable. If part of an algorithm can be controlled in a meaningful way then these configuration settings can always be inspected and changed by calling code.

3. All errors are reported via exceptions, nothing is printed to stdout by the framework. This produces the most meaningful error messages which can be handled by humans or machines, and produces cleaner command line applications.

4. Verify everything, this framework is intended to be extended by the user. As such each object will contain a function to verify its own integrity. The framework includes complex data structures such as tensor graphs and multidimensional indexing models. If manipulated incorrectly these structures can easily enter invalid states leading to difficult to trace bugs. These verification functions are a valuable debugging aid to find the root of cause of these bugs.

5. For maximum portability the generated code will have no dependencies beyond ANSI C built-ins. To support research on as many target devices as possible the most established and well supported language has been chosen with no external dependencies.

6. Dynamic memory allocation is not be used by the generated code. This is forbidden by the coding standards used in flight software in the space industry. Additionally many terrestrial micro-controller systems do not support this functionality.

7. The code generated is tailored to be optimised by the compiler. Compilers are very good at what they do, TFMin doesn't make any attempt to do their job for them. Higher level optimisation is performed by TFMin, lower level optimisation is left to the compiler.

4.3.2 Graph Representation

Core to the design of TFMin is a generic internal representation of an ML models tensor graph. This data structure includes a complete description of an inference model including; operations, weights, topology, inputs, and outputs. A $tf_min.Graph$ object is created using an importer from either a Tensorflow session or TFL flatbuffer. Unlike Tensorflow sessions these Graph objects are mutable and can be altered directly or passed through Graph-Translation objects. Multiple Graph-Translation objects can be chained together in a manner analogous to compiler optimisation steps. These translation pipelines allow great flexibility when optimising models and controlling how models are deployed using TFMin. Researchers can configure existing Graph-Translators and chain them together in different ways to deploy models or entirely new Graph-Translators can be derived from existing ones.

Alongside information which describes a model, $tf_min.Graph$ objects also store two types of information required for final deployment. A sequence of operations defines the execution order of the model and pre-allocated offsets place each intermediate tensor buffer within the tensor arena. Models do not include this information when they are initially imported but it is required by the final code generation step, so must be generated during the graph translation pipeline described in Section 4.3.3. Graph-Translators implementing a range of sequencing and memory allocation algorithms are provided.

Graph Object	
ops	List of Operations
Un-ordered list of operations contained in this tensor graph.	
tensor	List of Tensors
Un-ordered list of tensors contained in this tensor graph.	
op_sequence	List of Operations
Ordered list of operations containing exactly the same set as 'Operations' but in a valid execution order.	

Operation Object	
op_type	String
Defines the type of this operation.	
inputs	List of Tensors
Links to the input tensors of this operation.	
outputs	List of Tensors
Links to the output tensors of this operation.	
params	Dictionary
Set of parameter names and their values customising this operation.	

Tensor Object	
tensor_type	Enum
INPUT, INTERMEDIATE, CONSTANT or OUTPUT	
meta_type	Enum
SINGLE, SUPER or SUB	
d_type	String
Element data type of this tensor.	
shape	List of Int
The size of each dimension of this tensor, length of this list defines its rank.	
memory_map	MemoryMap
Object holding the data layout of this tensor.	
values	Numpy.ndarray
Optional attribute holding the value of this tensor if it is constant.	
memory_offset	Int
Optional attribute holding the pre-allocated offset within the tensor arena.	

Figure 4.2: Data structure of the three Core objects TFMin uses to represent ML models internally. In the interests of clarity only high level attributes are shown.

4.3.3 Graph Translation Pipeline

A single Graph Translator is in essence a function which takes a Graph object as a parameter and returns a new modified Graph object as a result. The graph translation pipeline is a sequence of these translators and their associated parameters. A complete pipeline is a powerful way to define how a model is optimised deployed to C code. Graph-Translators can be used to modify a models graph itself, sequence operations, pre-allocate tensor buffers or output introspection information without modifying the graph.

The TFMin API provides two methods to directly interact with Graph-Translators. Static function calls are the easiest method when a graph is being manipulated directly from python code. However Graph-Translator objects are implemented internally as functionoids [37] so their configuration parameters are decoupled from the translation call itself. This allows Graph Translator instances to be created, loaded, and stored as objects but also called as functions, both these calling patters are shown in Listing 4.1. Some Graph-Translators contain other translators, for example the operation-splitting memory optimiser uses a block-level memory optimiser within its algorithm.

Listing 4.1: Example GraphTranslator calling patterns.

```
class ExampleTranslator(GraphTranslator):
        def __init__(self, settings):
                defaults = {'setting_1': 5}
                super().__init__(settings, defaults)
        self.translator_type = 'ExampleTranslator'
        self.description = "Example null graph " \
                           "translation object."

        def __call__(self, input_graph):
                output_graph = input_graph.clone()
                return output_graph

# Static calling pattern
new_graph = ExampleTranslator.call(old_graph,
                                   params={'setting_1': 10})

# Functionoid calling pattern
fnoid_translator = ExampleTranslator({'setting_1': 10})
new_graph = fnoid_translator(old_graph)
```

Graph translation pipelines are manipulated and executed using the Pipeline object. This object allows the complete configuration of a pipeline and all Graph-Translators within it to be loaded and saved to Extensible Mark-up Language (XML) files. This object makes it simple to control and record the translations and their parameters which are applied when deploying models.

Using these features of the Graph Translation pipeline, the deployment of a ML inference model can be easily defined and recorded using XML files. Listing 4.2 shows how memory allocation performance was analysed while deploying models using the novel DMO algorithm proposed in Section 5.4.2. When performing high level research using the built in functions of TFMin, most work can be done by modifying these XML files.

Listing 4.2: XML TFMin configuration describing the deployment and introspection of a model that was used to evaluate the operation splitting memory optimisation technique on different models.

```
<tfmin>
  <pipeline>
    <RemoveIdentityOps />
```

```
    <RemoveDropOuts />
    <SequenceOps execution="greedy" />
    <ExportGraph filename="sequenced_graph.svg" />
    <OperationSplitting>
      <block_allocator>
        <HeapSmartOrder order="forwards" />
      </block_allocator>
    </OperationSplitting>
    <ExportGraph filename="op_split_graph.svg" />
    <ExportMemoryLayout filename="memory_map.svg" />
  </pipeline>
  <code_gen name_prefix="test_model" output_base="test_" />
</tfmin>
```

4.3.4 Using Graph-Translators for Introspection

An additional use of the graph translation pipeline, is for introspection when analysing deployment experiments. Graph Translators are provided which instead of modifying the graph, export information about the graph for visualisation or processing outside of TFMin. The Graph and Memory Visualisers, generate vector images of the Graph structure and pre-allocated buffer pattern respectively, Figure 4.3. To emphasise parts of models, these visualisers take advantage of the highlight attribute contained in each tensor and operation object. The highlight attribute can be either None or a Red Green Blue (RGB) tuple. Algorithms within Graph-Translation objects can set these attributes allowing clear visualisation of their internal workings. By placing these visualisers at points along a deployment pipeline, detailed behaviour of the deployment can be observed.

4.3.5 Tensor Memory Model

The memory used to hold tensor buffers is managed differently depending on of the type of tensor, input/output, constant weights, or intermediate value. Buffers used for input and output tensors are allocated by the calling process, allowing their memory to be allocated optimally for the application. This could include using specific Direct Memory Access (DMA) memory banks for the input tensors or reusing the same output tensor buffer for example. The data layout of these tensors is defined by the imported model, which is limited to row-major or column-major

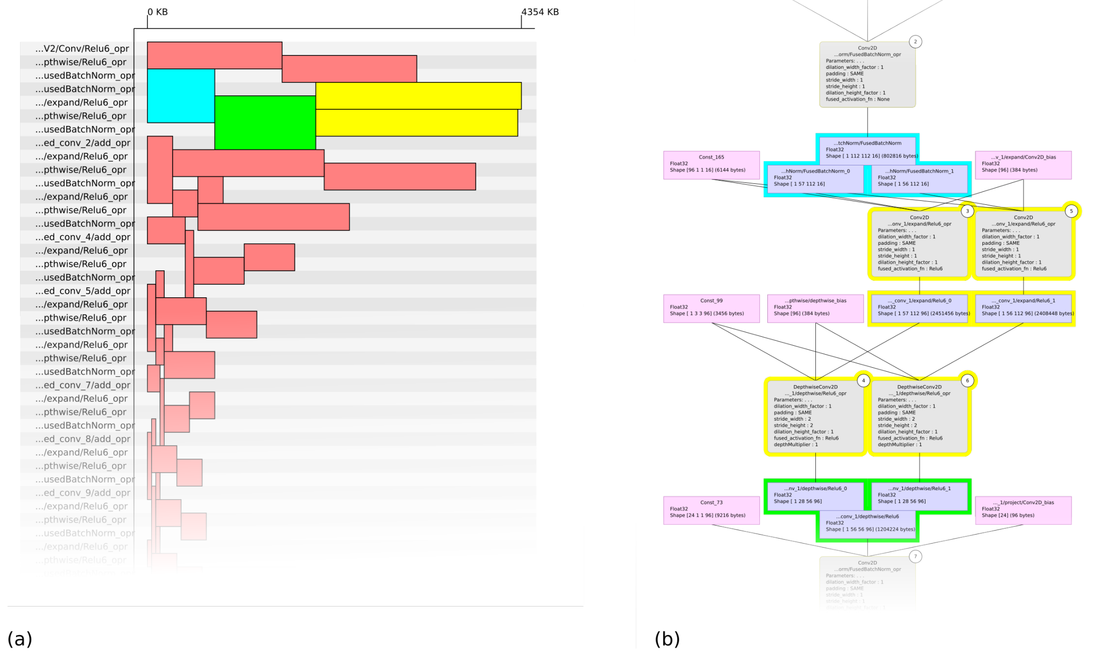


Figure 4.3: Example of (a) memory and (b) graph visualisations generated generated after operation-splitting optimisation, see Section 5.4.1. Operation and Tensor highlights have been added by the algorithm to clearly identify the parts of the model's graph which were split into parallel chains.

by the supported input formats.

Weight tensors are provided by the imported model and these constant values are currently built into the compiled binary by TFMin. This is one area that could be extended to support more weight storage options. Embedded systems could store this type of data in a memory mapped Electronically Erasable Programmable Read Only Memory (EEPROM), while larger computers could load them from a file system or network. Weight compression schemes [83] [61] could also be supported by custom weight exporter objects. The values of weight tensors cannot be changed during the deployment process however their data layout can be. A common optimisation technique changes the layout of convolution filter weights so that they are read from sequential memory addresses by the operation code [92].

Intermediate tensors are used to hold the value transferred between a models high level operations, these tensors are often only required for a fraction of the whole inference process.

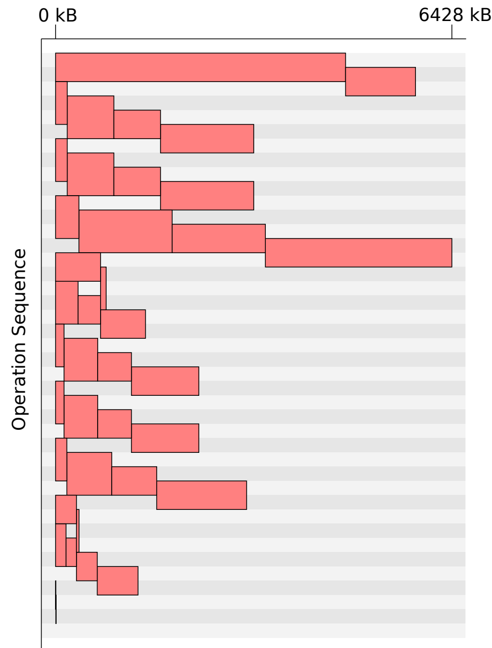


Figure 4.4: SqueezeNet intermediate buffer layout within a 6.3 MB tensor arena. Buffer locations computed using a forwards pass of a heap algorithm.

TFMin stores these tensors in a single contiguous memory area known as the tensor arena, their locations within this area are pre-computed during the deployment process. Figure 4.4 shows the intermediate tensor locations for the SqueezeNet model pre-allocated using a heap approach, the visualised scope of each buffer is used to reuse the memory of the tensor arena efficiently. Computation of optimal intermediate tensor layouts within this tensor arena is a new area of research, which TFMin has been designed specifically to support. This research has yielded two new state of the art memory optimisation algorithms DMO and Operation-splitting described in Chapter 5.

A requirement which was added to TFMin during our later research into memory optimisation, sets it apart from both TFL and μTensor. The capability to store intermediate tensors using non-contiguous in memory representations as well as dense row/column major representations. This requirement was driven by research into the operation-splitting technique described in Section 5.4.1. In this work sub tensors produced or consumed by layout operations such as split or merge are able to use the memory space of the large tensor instead of a separate region in the tensor arena. Figure 4.5 shows a simple split operation which requires output tensors occupy their own areas of memory when dense memory addressing is used. Whereas if non-contiguous

memory indexing is used, input tensor memory can be re-used without copying any data. In this case the split operation disappears entirely.

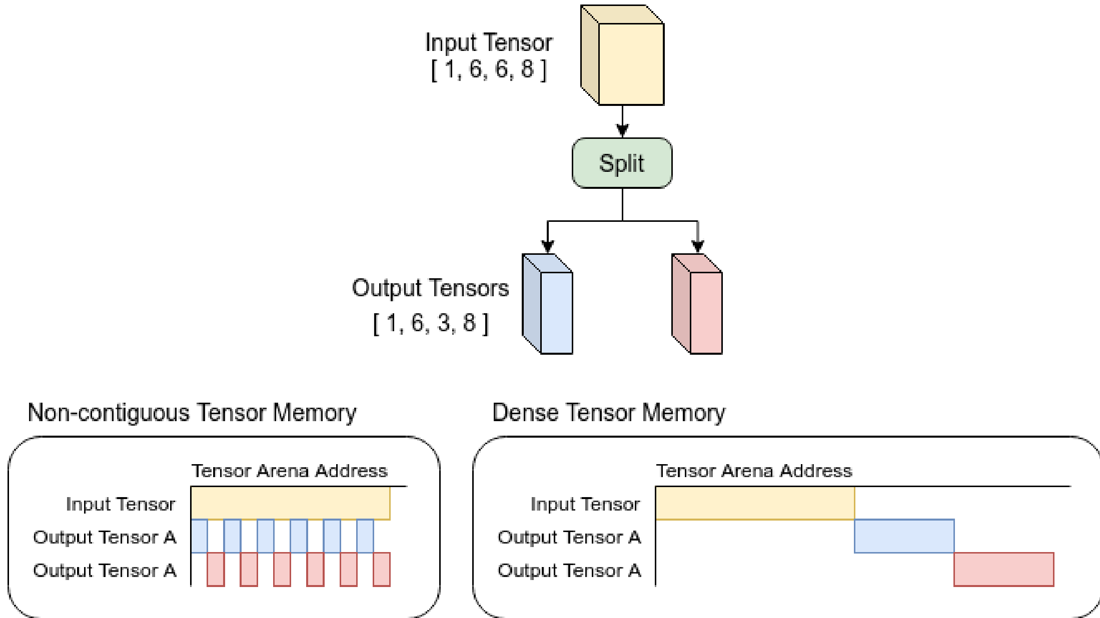


Figure 4.5: Example of tensor-buffer reuse which is made possible by non-contiguous memory addressing.

Row major, column major and non-contiguous layouts are in essence functions which convert a multi-dimensional address into a flat one dimensional address. Equations 4.1 and 4.2 define these functions for N dimensional tensors using row and column major layout schemes respectively.

$$Index = i_{N-1} + D_{N-1}(i_{N-2} + D_{N-2}(i_{N-3} + D_{N-3}(\ldots + D_1 i_0)\ldots))) \tag{4.1}$$

$$Index = i_0 + D_0(i_1 + D_1(i_2 + D_2(\ldots + (D_{N-2} i_{N-1}))\ldots)) \tag{4.2}$$

Where N is the rank of tensor being indexed, $i_0 ... i_{N-1}$ are the indices within each dimension, and $D_0 ... D_{N-1}$ are the sizes of the tensor in each dimension.

The indexing function for non-contiguous layouts is shown in Equation 4.3 where C is a constant defining the offset of the first element from the start of the tensors memory and $S_0 \ldots S_{N-1}$ are the index strides between adjacent slices of each dimension. Mathematically C is not strictly

required, however non-contiguous arrays can index backwards and it is good practice to avoid negative indices.

$$Index = i_0 S_0 + i_1 S_1 + i_2 S_3 + \ldots + i_{N-1} S_{N-1} + C \tag{4.3}$$

Expanding the row and column indexing Equations 4.1 and 4.2 gives Equations 4.4 and 4.5. It can be seen that the non-contiguous indexing Equation 4.3 can generalise all indexing schemes by defining the strides as the products of the relevant sets of dimensions. This reasoning is behind the choice to describe all tensor layouts within TFMin with an object which describes non-contiguous array indexing, this object contains two functions which detect the two special cases of row and column major ordering.

$$Index = i_0(D_1 \cdot D_2 \cdot \ldots \cdot D_{N-1}) + \ldots + i_{N-2}(D_{N-2}) + i_{N-1} \tag{4.4}$$

$$Index = i_0 + i_1(D_0) + i_2(D_0 \cdot D_1) + \ldots + i_{N-1}(D_0 \cdot D_1 \cdot \ldots \cdot D_{N-2}) \tag{4.5}$$

There are several drawbacks of non-contiguous array representations which is why they are not used in high performance applications. They often force sequential read and write operations to occur on non-sequential memory addresses which is detrimental to cache performance. They also make the implementations of optimal tensor operations more complex to write. Both of these drawbacks are mitigated in TFMin because it is a code generator, this gives it complete control of how tensors are represented and the versions of operation kernels which generate the code which implements each operation.

4.3.6 Operation Kernels

At the core of TFMin is the generation of tensor operations in C code, this functionality is contained within a set of Operation-Kernel objects. Each tensor operation is supported by a matching Operation-Kernel. During the final code generation step operations are processed sequentially and appropriate Operation-Kernels found and invoked to produce the C source.

Listing 4.3: TFMin generated implementation of two layer dense model.

```
void model(void *tensor_arena,
           float *const p_input_x_input_0,
           float *p_Layer2_activation_0) {
  /* MatMul op (Layer1/Wx_plus_b/MatMul) */
  {
    const float *input_0 = p_input_x_input_0;
    const float *input_1 = (float *)Layer1_weights_Variable_0;
    const float *input_2 = (float *)Layer1_biases_Variable_0;
    float *output_0 = (float *)tensor_arena + 0;
    for (int b = 0; b < 1; ++b) {
      for (int out_c = 0; out_c < 300; ++out_c) {
        float value = 0.0f;
        for (int d = 0; d < 784; ++d) {
          float inputVal = input_0[(b * 784) + d];
          float weightVal = input_1[out_c + (300 * d)];
          value += inputVal * weightVal;
        }
        // Add Bias
        value += input_2[out_c];
        // fused activation function
        // Relu
        if (value < 0.0f) value = 0.0f;
        output_0[out_c + (300 * b)] = value;
      }
    }
  }
  /* MatMul op (Layer2/Wx_plus_b/MatMul) */
  {
    const float *input_0 = (float *)tensor_arena + 0;
    const float *input_1 = (float *)Layer2_weights_Variable_0;
    const float *input_2 = (float *)Layer2_biases_Variable_0;
    float *output_0 = p_Layer2_activation_0;
    for (int b = 0; b < 1; ++b) {
      for (int out_c = 0; out_c < 10; ++out_c) {
        float value = 0.0f;
        for (int d = 0; d < 300; ++d) {
          float inputVal = input_0[(b * 300) + d];
          float weightVal = input_1[out_c + (10 * d)];
          value += inputVal * weightVal;
        }
        // Add Bias
        value += input_2[out_c];
        // fused activation function
        // None
        output_0[out_c + (10 * b)] = value;
      }
    }
  }
}
```

Listing 4.3 shows the code produced for a simple two layer fully connected model. The sizes and layouts of tensors are static, so their locations can be encoded within each operation. The resulting model function is a set of code blocks for each operation which only interconnect via the tensor arena and input/output buffers. These code blocks avoid conflicts between the identifiers of different layers. An advantage of the code generation approach is that the sizes and parameters of layers are compile-time constants, enabling further compiler optimisation. This would not be possible if a library of tensor operations were being used. The downside of this approach is that large models result in large binaries being generated. Investigating the trade-off between binary size and execution speed is a potential future area of research.

The common Operation-Kernel interface allows kernels to define a set of tag strings describing any special capabilities they have. For example the authors have developed two Operation-Kernels optimised for the LEON3 using the MAC instruction supported by the processor. The assembly statements generated by these kernels will not build on other targets, so they are tagged for the ”LEON3”. When a model is deployed the ”LEON3” tag can be specified which will select these custom Operation-Kernels where possible while falling back to the standard kernels for most operations.

The set of Operation Kernels objects can be used for more than just their primary purpose of generating code. They are a link between the theoretical operations within Graph objects and their ultimate implementation on hardware. This link is exploited by the DMO memory optimiser which overlaps the input and output buffers of operations. The extent by which these buffers can be safely overlapped is defined by the final layer implementation deployed. This memory optimiser uses the Operation Kernel system along with the mutable nature of python objects to add functions into kernels which calculate the safe overlap for a given operation.

4.3.7 Memory Optimisation

Reducing the RAM required to perform ML inference on small hardware targets has been the aim of much of the research performed using TFMin to date. As described earlier all intermediate tensor values are stored in a contiguous memory area known as the tensor arena.

Before the final code generation step the location and data-layout of all intermediate tensors within the tensor arena must be defined. A set of Graph-Translators are provided by TFMin which encapsulate the novel memory optimisation algorithms produced during my research as well as pre-existing algorithms.

The simplest class of memory optimisers are block-level algorithms, such as the heap allocator used to pre-allocate the tensors buffers for SqueezeNet shown in Figure 4.4. These optimisers take a sequenced graph and use the known sizes and scopes of each intermediate buffer to place them within the tensor arena. The locations of these tensor buffers are chosen to minimise the total size of the tensor arena. At first glance this sounds like a simple task, but on closer inspection it is NP-Hard [95].

There are similarities between this task and the well studied register allocation problem [87]. Both are concerned with the minimisation of intermediate storage required to evaluate a computation graph. Registers however are fixed size atomic entities, whereas the tensors used in ML graphs are variable sized objects which can be represented in different ways and divided into parts. Within the register allocation problem the cost of re-computing a value is often less than temporarily storing it in memory. This means that re-computation is common technique used to make optimal use of available registers. However the cost of re-computing full tensors is prohibitive on the low power CPUs this work focusses on. These differences mean that the existing algorithms which solve the register allocation problem are not applicable to the reduction of acram used for ML inference.

One of the mathematical tools used in the register allocation problem does find some application to this problem though. The Strahler number [39] of a computation graph can be used to determine the minimum number of registers needed to complete the computation without any re-computations. However this is only applicable to models whose computation graphs take the form of trees, which is rare for current state of the art models. In the memory optimisation task the Strahler number of a tensor tree determines the minimum number of intermediate tensors which are needed at any one time. However since intermediate tensors have different sizes, reaching this number does not guarantee optimality

Looking closely at the SqueezeNet buffer layout shown in Figure 4.4 it can be seen that the heap allocator has not produced an optimal layout. Figure 4.6 shows how this could be optimised

further, heuristic algorithms are able to improve results but will always be limited by a lower-bound limiting how small the tensor arena can be. For block level optimisers this lower bound is the largest sum of buffer sizes whose scopes overlap. In the example shown in Figure 4.6 it can be seen that this lower bound has been reached, all three tensors buffer which were shifted left need to be in memory at the same time. So the smallest tensor arena size possible is the combined size of these three buffers.

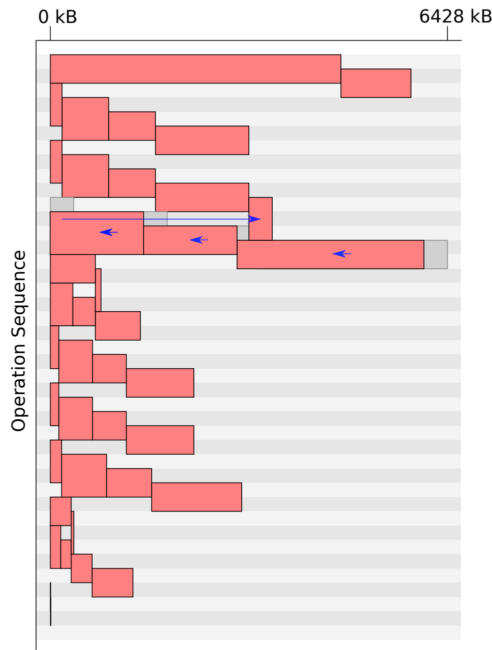


Figure 4.6: SqueezeNet intermediate buffer layout, showing how the layout generated using a heap algotithm shown in Figure4.4, can be further optimised to use less memory.

This limitation can be surpassed by the next two classes of memory optimiser. Graph-level optimisers alter the tensor operation graph itself or the sequencing of operations in order to improve the result produced by a block-level optimiser. A simple example would be a eager/lazy sequencer which tests the peak memory in each case and chooses the best. A more complex example would be the operation-splitting optimiser described in Section 5.4.1.

Finally memory-access-level optimisers analyse the precise memory access patterns of each tensor operation attempting to overlap the buffers of the inputs and outputs of single operations. DMO described in Section 5.4.2 is currently the only example of such an algorithm the author is aware of. Chapter 5 discusses the details of these algorithms in more detail.

4.4 Analysing Deployed Models

As would be expected of a research support tool TFMin can produce a wide range of observations of both the generated inference model and the deployment process that led to them. These include simple metrics such as the size of models weights, RAM requirement and per layer execution time measurement. More complex introspections are available to observe the behaviour and results of memory pre-allocation algorithms and detailed memory use of final deployed binaries. A Verification tools is included which compares the output of a deployed model to the output of its original implementation in Tensorflow. These tools and examples of the observations they produce are described in the following sections.

4.4.1 Memory Requirements Analysis

When an inference model is deployed the size of the tensor arena is a single scalar, which is a measure of the efficiency of its memory use. There is little analysis which can be done on this single number, beyond comparing different optimisation algorithms. However to understand the processes that led to a deployed model having a particular tensor arena size it is useful to visualise the pattern in which intermediate tensor buffers have been allocated. This analysis can be performed by inspecting the buffer pre-allocation pattern as it is generated and refined during the deployment.

This introspection can be generated using the memory visualisation graph translator as described in Section 4.3.3, the default behaviour of this visualiser highlights the buffers which directly define the tensor arena size, Figure 4.7. This visualisation is particularly useful for complex memory optimisers such as operation-splitting, Section 5.4.1, where block level optimisers are repeatedly executed on modified graphs searching for the most optimal solution.

It is interesting to note that many times during our research into memory optimisation algorithms when the resulting buffer allocation patterns have been observed, potential improvements have been clearly visible. This is testament to both the mathematical complexity of the problem and humans innate skill at solving packing problems. The pattern shown in Figure 4.7 can be improved by swapping the order of buffers 1 and 2 in the memory dimension, this is one of many optimisations which could be made in this case.

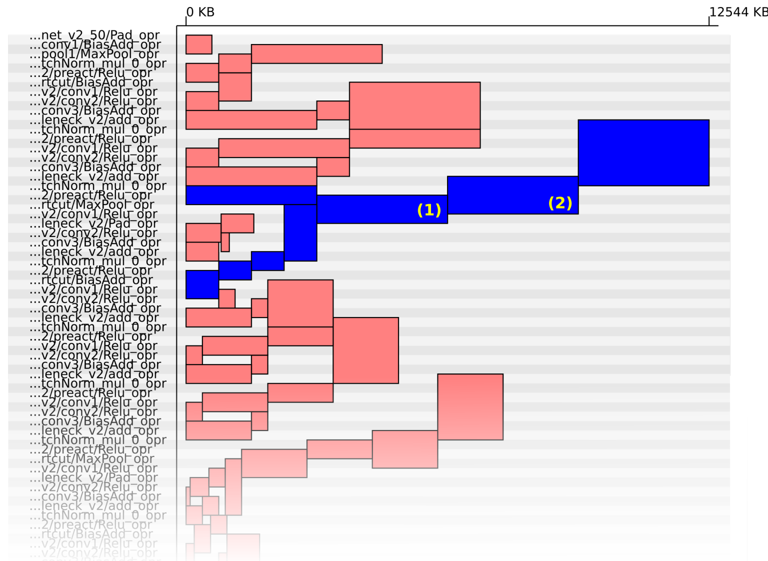


Figure 4.7: Pre-allocation pattern for a deployment of the Inception Resnet V2 model. Buffers which define the tensor arena size are highlighted in blue.

4.4.2 Detailed Memory Access Analysis

During our research into the memory use patterns of deployed ML models and available optimisation techniques, the exact pattern of memory accesses issued by a compiled inference model needed to be observed. After searching for any such tools none were found that could extract the information needed. This section gives a technical overview of the Visual Memory Tracer [24] tool that was built and how it is used by TFMin to perform introspection on compiled ML inference operations.

Our goal was to determine which areas of memory were holding values used by the computation and which were not as a model was executed. To gain this information is was necessary to observe the pattern of load and store instructions issued by an executed model, then analyse which regions of memory were in use as inference progressed. Memory access events needed to be recorded in a theoretical 2D time/address space and aggregated to a lower resolution in both dimensions. Even medium sized ML models will issue hundreds of millions of load and store instruction.

It should be noted that this memory use pattern can also be found by analysing tensor operation source code directly. This approach was used during later research into efficiently implementing

automatic DMO in Section 5.4.3. However during earlier stages of this research the memory use of models deployed using a range of tools was investigated, TFLμ, μTensor, and TFMin. It was simpler to use a single tool to analyse executed binaries than hand analyse a large number of tensor operation implementations. There was also the risk of mistakes in manual analysis of unfamiliar code impacting the validity of research conclusions. This risk does not exist when directly observing the behaviour of a compiled binary.

Building a Memory Tracer Using Valgrind Lackey

Valgrind [111] is a powerful suite of debugging tools which aids the discovery memory leaks and other memory related bugs, it contains a simple tool called 'Lakey' which prints the address and type of every memory access operation issued by the binary on test to the standard out stream. This output provides all the necessary information to produce the aggregated memory trace required, it does however produce huge amounts of raw data. Initially a tool was written to process this raw stream of memory events into a useful visualisation of memory use, this tool later evolved into the stand-alone VMT tool.

The first version of this tool recorded the memory accesses for the whole duration of execution but only the memory space used for tensor operations, the tensor arena. This requires the binary on test to pass the address and size of the tensor arena to the memory tracer tool, while analysis is in progress. A linux First In First Out (FIFO) is used to communicate between these two processes. Three binaries are executed for each analysis performed using the Visual Memory Tracer, the connections between them are shown in Figure 4.8. A client library is provided by VMT to encapsulate the required communication between the binary on test and the tracer tool, this handles management of the FIFO and provides a clean interface for the developer.

The raw output from the lakey tool describes every load, store, or modify instruction, it is possible to decimate these events into fewer aggregated events. However the meaning of this aggregated data needs to be carefully defined. This decimation groups all memory access instructions which address a region of memory for a period of time into a single read/write/modify event. A read event is defined as any set of memory accesses where an address is read from before that exact address has been written to. A write event is defined as any set with at least one write or modify access, if both these statements are true then the set is aggregated to a modify event.

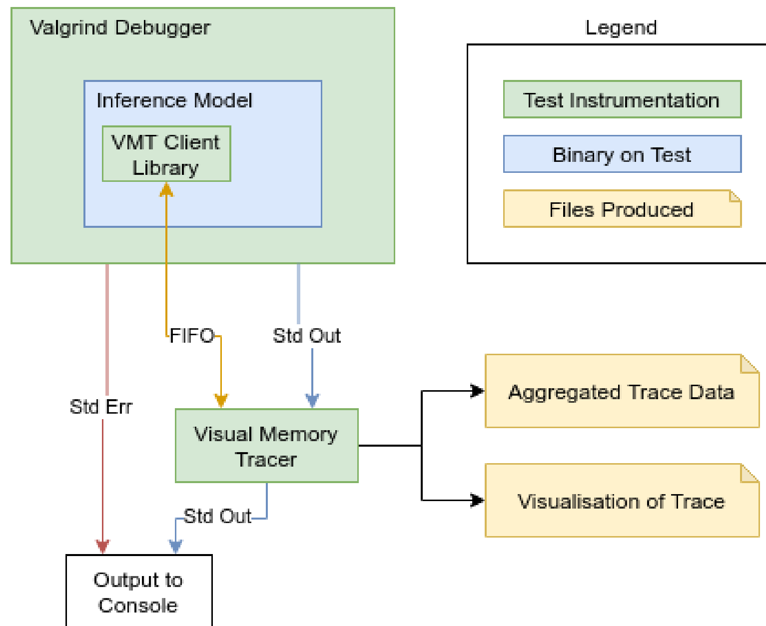


Figure 4.8: Information flow diagram of a binary being traced by the Visual Memory Tracer.

The array of aggregated read/write/modify events recorded by the Visual Memory Tracer are used to identify areas of time/memory space which are not used. Defined as areas which are either not read from in the future or are written to in the future before being read later. This was the critical information needed during our research because these unused areas indicate where there is potential to use memory more efficiently, Figure 4.9.

It is important to note that this aggregated data will be meaningless if the pattern of memory access being observed has details smaller than the resolution used. The ML models analysed are made of tensor operations which access large buffers often taking millions of instructions to execute. So as long as the resolution of the data in time and memory is high enough to clearly represent the areas of time/memory space an operation occupies then the results will be meaningful.

Figure 5.6,a in Section 5.4.2 shows a trace plot of inference performed with the MobileNet v1 model. Here the white (unused) and grey (used) areas of time/memory space clearly indicate there is scope to used memory more efficiently. This research insight was the key which led to the creation of the DMO technique which can be used to significantly reduce the memory requirement of inference using ML.

TFMin integration with the Visual Memory Tracer

The current version of the Visual Memory Tracer tool which I have made available on GitHub supports signalling the start and end of events by the binary on test. TFMin can now automatically add the VMT client library and code needed to trace any model using this tool, including highlighting the scope of each operation in time and the location of pre-allocated buffers in time/memory space. This feature allows the detailed observation of the memory use pattern of deployed models as shown in Figure 4.9.

4.4.3 Analysing the Output of Generated Implementations

Alongside code generation, TFMin provides a set of tools to evaluate the performance of generated model implementations. These tools use the capability to generate, build, execute, and test ANSI C implementations encapsulated within the $GraphEvaluator$ object. The capability to feed test data through a deployed model using python calls can be used for a range of different tests, analyses, and even model re-training. The simplest analysis builds and runs a model while measuring the execution time of each layer. Full analyses run the deployed model feeding test data through it and comparing the output to another version of the model, this can be either a different TFMin implementation or from a different framework entirely such as Tensorflow.

Unit Testing

During the development of TFMin as with all good software it was necessary to build a suite of tests to validate the code as it is built and maintained. These tests include unit-tests of the OpKernels which generate c implementations of individual layers. To validate a code generator it is necessary to validate the generated code. This is performed by comparing the output of the deployed model to the output of the same model executed in the Tensorflow framework.

In the case of floating point operations determining if these tests are passed or not is non-trivial and upon closer inspection it is revealed that we are interested in two different types of failure. The first type of failure is a simple bug in the implementation causing gross errors in the output. The second type of failure is more subtle, changes in the numerical instability of layers causing a model implementation to perform below the accuracy achieved during training.

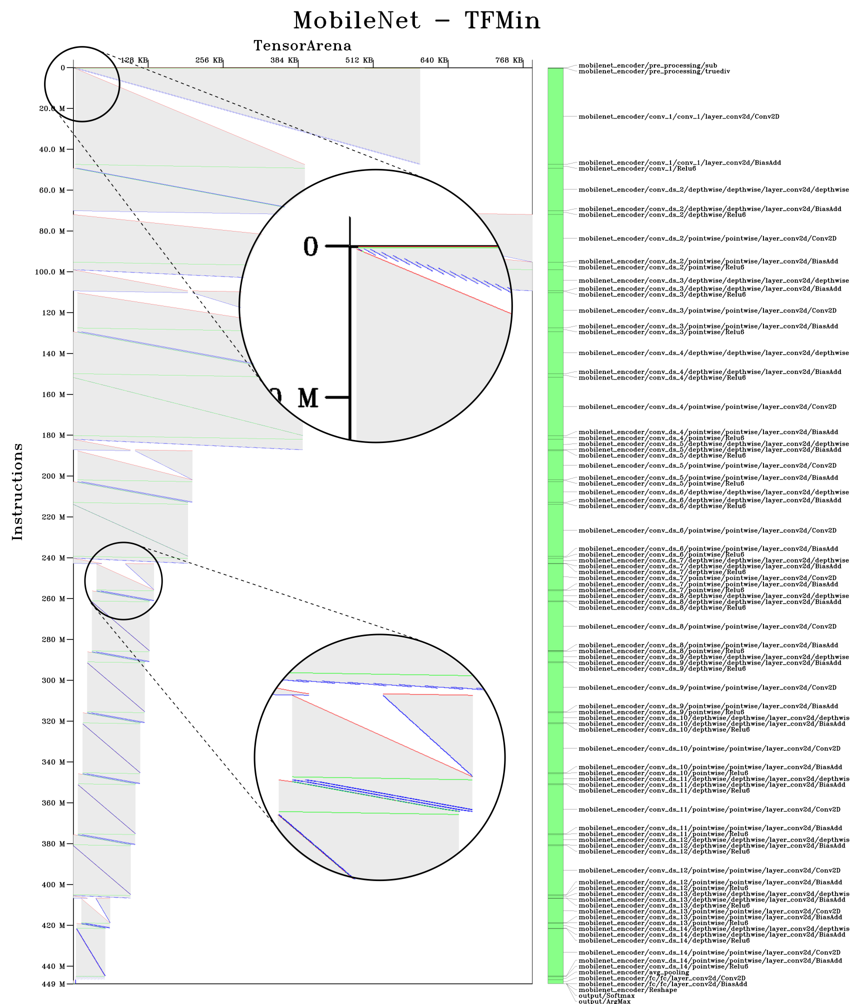


Figure 4.9: Full memory trace of a DMO optimised deployment of Mobile Net using TFMin. High resolution images are produced by VMT to better capture the fine detail of the memory trace. These do not work well in print, so zoomed areas are provided to reveal the details of the tensor operations.

Unlike atomic scalar Floating Point Operation (FLOP)s, tensor operations are compound operations made up of many FLOPs and there are different mathematically valid orderings which nonetheless produce different results due to floating point rounding errors. Implementations using faster MAC instructions will be rounded differently to implementations using atomic multiplication and addition instructions [151]. Dense layers require the summation of large numbers of floating point values, the particular implementation of this summation can have a significant effect on the numerical stability of its output. [68]. This second type of failure

is characterised by the same model with the same weights and same input data producing meaningfully different output when executed by two different implementations.

It is debatable if failures due to increased numerical stability should be unit-test failures or not, given that machine learning is a branch of inexact computing. In the case of TFMin it was decided that these problems are not checked during unit-testing, but would instead be addressed using model re-training as described in Section 4.4.3.

Model Re-Training

It has been discussed that the low level implementation of floating point models can effect the accuracy of inference. There are a wide range of potential causes of these differences. The numerical instability of the algorithms used can alter the precision of layer outputs, any change from the algorithm which was used during training can negatively effect inference accuracy. Both increases or decreases in numerical stability can cause these problems, since it is the difference in the algorithms used for training and inference which introduces the change. Compilers and hardware can also introduce these problems even when layer code is identical. Certain embedded Floating Point Unit (FPU)s do not adhere to the Institute of Electrical and Electronics Engineers (IEEE) 754 standard, for example the LEON3 FPU does not support sub-normal numbers instead rounding these values to zero. While compiler optimisations such GNU Compiler Collection (GCC)s -Ofast enable un-safe floating point simplifications, allowing the compiler to re-order FLOPS in ways which can alter numerical instability.

These causes all produce differences in the final output of a model, which will be larger the more layers a model has due to accumulation of errors. Although it is theoretically possible to produce an implementation with exactly the same numerical stability as another, often these changes are caused by desirable optimisations.

It is proposed that a re-training step which uses the output of a deployed model could reverse the reduction in model accuracy. Time has not permitted this technique to be invested during this work, so it is recommended as a future area of study.

Verifying the Correctness of Graph Translators

TFMin has been used to study two complex memory optimisation techniques, both Operation-Splitting (Section 5.4.1) and DMO (Section 5.4.2) make low level changes to the way a model is executed. In the case of both optimisers implementation bugs or flaws in the underlying theory would result in inference implementations which do not correctly execute the model.

Verification that these algorithms were not mathematically altering the model, was performed using TFMin to compare the results of the original and optimised models. In this case no difference in numerical stability is expected since identical layer code and compilers were used, so these models were expected to produce identical results. The $DeploymentComparison$ object performs this check using just three lines of code as shown in Listing 4.4. First the object is instantiated with both input graphs, then each model is deployed and executed with identical test data. The $Results$ object produced includes the complete results of each deployment, but in this case only the $max_error()$ method is needed to check if they are equal.

Listing 4.4: Example verification which tests to graphs produce identical results.

```
test = DeploymentComparison(graph_a, graph_b, c_flags='-O3')
results = test.execute(test_input_list)
if results.max_error() == 0:
        print('Graphs A and B produced identical results')
else:
        print('Graphs A and B produced different results')
```

Simple tests such as these are used in the unit-tests of all $GraphTranslator$ objects as well as during our research into the memory optimisation techniques described in Chapter 5.

4.5 Use Cases

The following three use cases describe how TFMin has facilitated our research during this PhD.

4.5.1 Layer Implementation Performance Analysis

During our early research deploying ML models onto the LEON3 processor, the first release of TFMin was used to compare the performance of a range of models on the LEON3 processor and a native Intel i7 processor. This version of TFMin produced C++ 11 code which depended on the Eigen Linear Algebra library [57]. This work was presented at the 13th Adaptive Hardware and Systems conference [22]. Five different ML terrain assessment models were analysed, Figure 4.10, in terms of their execution time on a 50 MHz LEON3 processor.

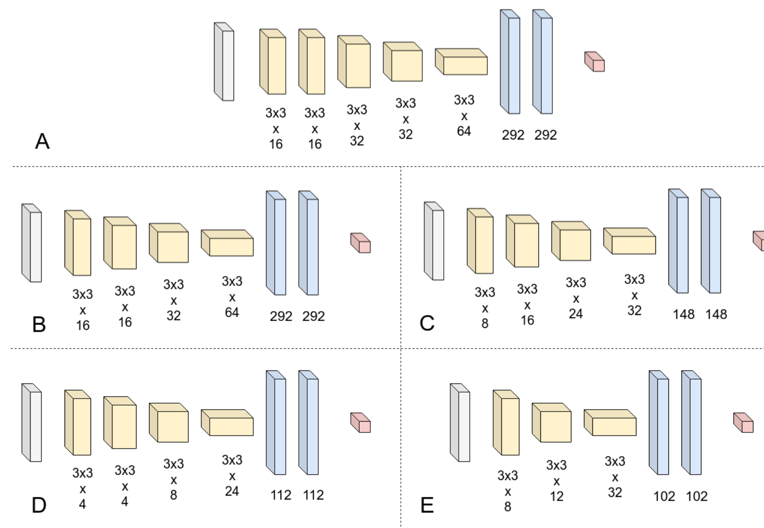


Figure 4.10: The five terrain assessment models analysed in [22], convolution layers shown in yellow and fully-connected layers shown in blue. Filter/weight sizes indicated under each layer.

TFMin facilitated this work by automating the deployment and performance analysis of models from the same python code which was responsible for building and training the models. Allowing more time to be spent tuning models and hyper-parameters.

4.5.2 Memory Optimisation

The analysis of novel memory optimisation techniques described in Chapter 5, required deployment features not supported by other Edge ML frameworks. Operation-splitting required that tensors support non-contiguous memory indexing while DMO required that the memory pre-allocation step is aware of the exact implementations used to execute each layer of the model.

Both techniques required that the locations of intermediate tensor buffers were pre-allocated during deployment.

The Operation-splitting memory optimisation technique was implemented in TFMin by a custom $GraphTranslator$ object. This translator alters the tensor graph of the model being optimised, Section 5.4.1. As well as requiring non-contiguous indexing the implementation and testing of Operation-splitting made use of the self-validation features of the TFMin framework.

The Operation-splitting technique alters a graph in such a way that a second downstream block level memory optimiser produces a better result than it would have done on the original graph. Therefore to analyse the performance of Operation-splitting it needs to be performed in combination with a range of conventional memory optimisers. Using TFMin it was possible to evaluate its performance on 23 different models using four block level optimisers in a matter of minutes.

Experiments demonstrating the effectiveness of DMO required less code than Operation-splitting but affected more disparate parts of the TFMin code-base. DMO is essentially a block level optimiser which is aware of the safe overlap between the input and output buffers of each operation. The size of this overlap depends on the operation type, parameters, and implementation used to compute it. For this reason the overlap computation was added to selected Op-kernels ensuring it was tightly coupled to the layer implementations themselves. Existing block level optimisers available in TFMin were extended to use this safe overlap information to produce the final DMO memory pre-allocations.

Unlike Operation-splitting DMO required a small extension to the architecture of TFMin, but it was cleanly implemented in the framework. Built in analysis and verification tools were used to confirm that the computed buffer overlaps were indeed safe. Introspection tools produced memory use plots such as Figure 5.6 which visualised the denser and more efficient and dense use of memory this technique achieves.

4.5.3 Computationl Requirements of Cost-Mapping Models

During the study of terrain assessment models proposed in Chapter 3 a crude and non-representative performance metric was used. Now that a deployment tool is available which can deploy and

analyse these models on the LEON family of processors more precise and representative results can be collected. This analysis was performed using a customised version of TFMin which is able to cross compile and remotely execute models.

The deployment analysis tools described in Section 4.4 were extended by the development of a custom $LeonRunner$ object, which builds and executes models on emulated or actual LEON processors. In our experiments models were built using the Gaisler LEON cross compiler and executed using the GRMON debugger, which uploaded and executes binaries on a GR-CPCI-GR740 Quad-Core LEON4FT [51] Development Board. Close integration allowed experiments and analysis to be run directly from the python environment even though deployed models were running on an external processor with a different architecture.

As well as the creation of a custom $LeonRunner$ object a new $GraphTranslator$ object was required. As mentioned the LEON3 FPU does not support sub-normal values, it never produces these values however if they are loaded to an FPU register an exception is thrown. To avoid this error a $SubNormalFilter$ object was created which rounds sub-normal float weights to zero or the smallest normal floating point value.

Our experimental use case demonstrates the utility of TFMin as a framework. Two new objects were derived from existing TFMin objects which extended the functionality of TFMin to analyse models on an external non-native target. It is worth noting that the byte order of the LEON3 is different to our desktop Intel computer, the switch to big-endian storage is handles automatically by the TFMin. This experimental set-up was used to produce the performance and accuracy results presented in Section 4.5.3. This extension to TFMin tool is owned by Airbus and is currently being developed into a commercial aerospace version of the tool.

Execution Time of Cost-Mapping Models

As expected inferences times on the LEON3 processor were significantly longer than those on a desktop GPU platform. Execution times were in the region of one thousand times longer which was not unexpected, however the relationship between model scale and execution time was also different. The GPU execution times in Figure 4.11 have an approximately linear relationship with model scale. There is also an offset of 2 milliseconds assumed to be an overhead of the TF framework. Execution times of the bare-bones implementation on a LEON3 processor

with FPU, and clocked at 200 MHz on the other hand are non-linear with no significant offset. This is what would be expected of a more accurate execution time measurement. There is no library start up overhead in the code generated by TFMin which is reflected in these timing results. The execution time of models at different scales follows a similar curve to the number of MAC operations in the model, Figure 4.12, which what would be expected of a single threaded implementation.

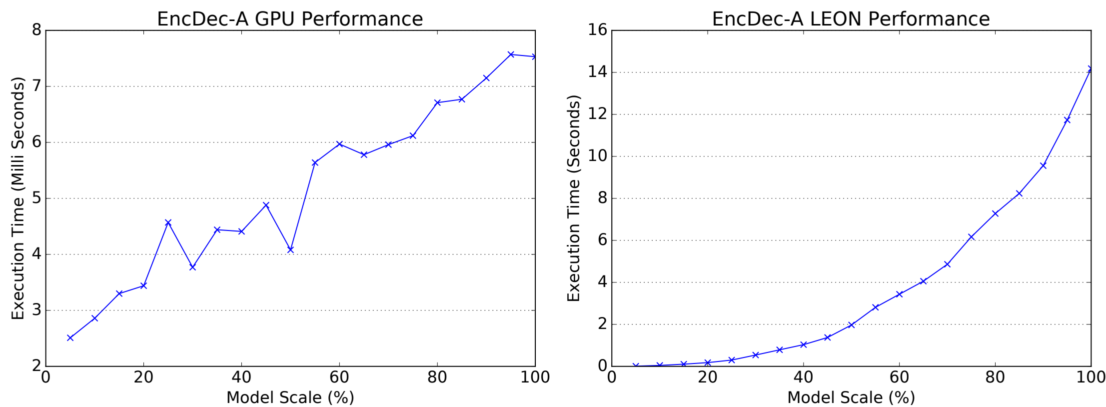


Figure 4.11: Inference execution time results of the EncDec-A model. Desktop GPU results shown in milliseconds on the left and single core 200 MHz LEON3 results shown in seconds on the right.

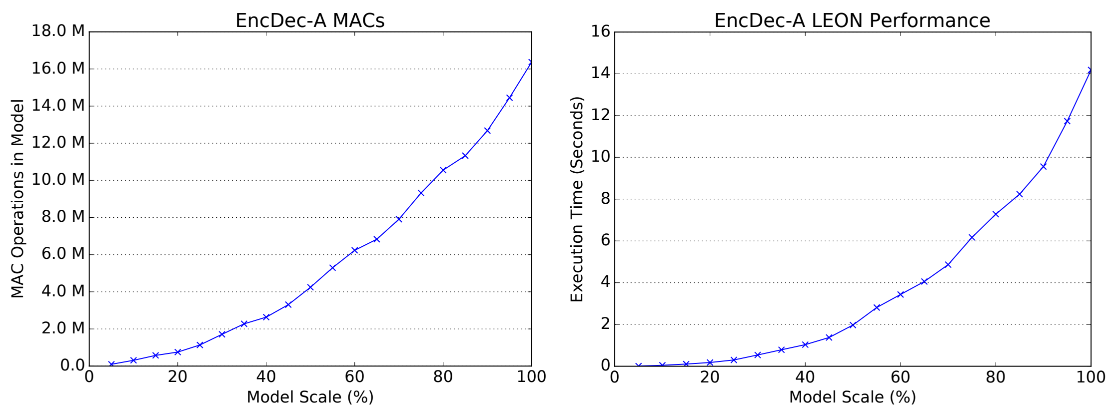


Figure 4.12: MAC operation count of the EncDec-A model compared to the execution time of this model on a LEON3.

Using the TFMin tool inference execution time results were collected for all scales of the cost-mapping models proposed in Chapter 3. These results were quicker to generate because the bare-bones LEON3 is completely deterministic, therefore models only needed to be executed

once. Unlike the desktop environment where the median of a large number of executions were taken to remove non-deterministic delays caused by the OS and other processes.

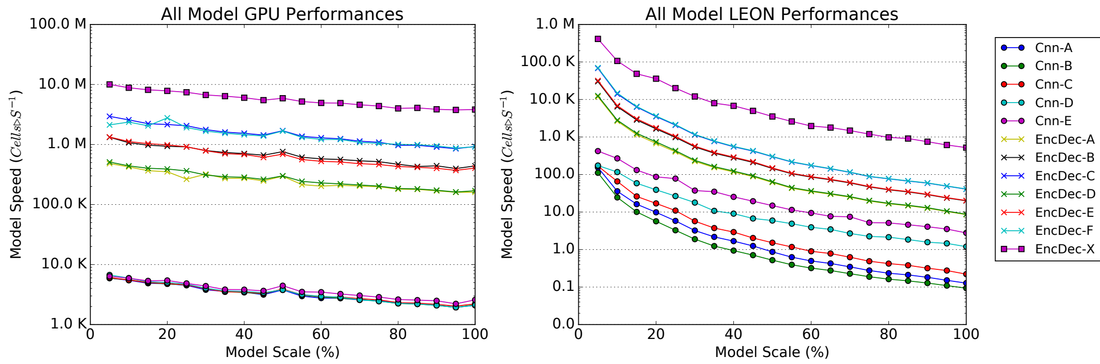


Figure 4.13: Inference execution time results of all cost-mapping models. Desktop GPU results shown on the left and single core 200 MHz LEON3 results shown on the right.

Figure 4.13 compares the performance, measured in $Cell.S^{-1}$, of all twelve proposed cost-mapping models. The relative speeds of models were estimated to a certain degree by the GPU timing results, but that was as far as their usefulness extended. When compared to the real performance results on the LEON3 processor we see that the scale of the model actually has far more impact than previously measured. We hypothesise that the cause of this difference is the nature of the performance bottle neck of these two different computing systems. A single core LEON3 implementation will be performance limited by either the FPU throughput or the speed of accessing memory, both of which are linearly related to the model MAC count. The Nvidia GTX-1070 GPU has 1920 Compute Unified Device Architecture (CUDA) cores so its performance will not be bottle necked by the number of MAC operations but by the overheads of setting up each tensor operation. Limitations of using a GPU implementation to time these models were expected and motivated the development of the TFMin tool.

Now that the true impact of model scaling on execution is known, the benefit of reducing the size of a cost-mapping model can be seen to have an effect of a comparable magnitude to increasing the number of cells produced in a single pass. The performance improvement between the Encoder-Decoder models F and X is approximately one order of magnitude, while the performance improvement between a full scale model and a 5% scale model is approximately three orders of magnitude across all models. This is an important finding because it implies that if the estimation errors of smaller models can be reduced below the requirement threshold then

a significant improvement in execution speed will result.

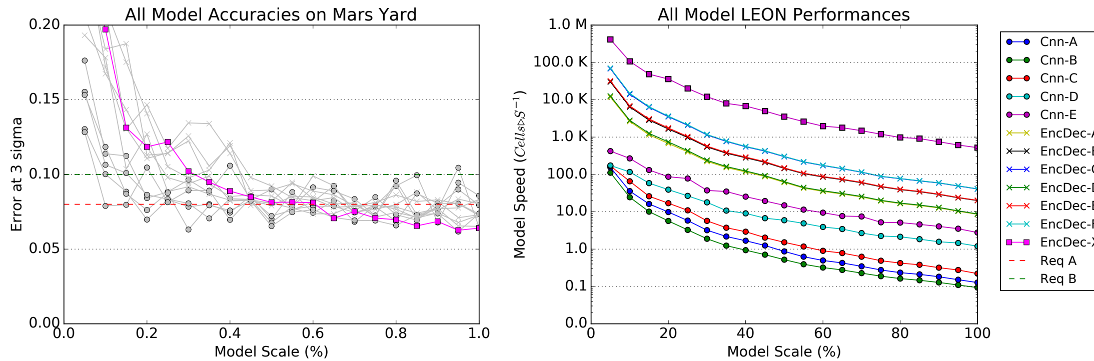


Figure 4.14: Accuracies of all models with the Encoder-Decoder model X highlighted shown in comparison to the performance of all models on a single core 200 MHz LEON3.

For example if there was a hypothetical error requirement of 8% at 3 sigma instead of our actual requirement of 10% at 3 sigma. Applying these to the highest performing model Encoder-Decoder X, Figure 4.14, results in a model scale of 0.65 satisfying our actual requirement, and a scale of 0.35 satisfying the hypothetical requirement. The performance of EncDec-X at these scales is 1783 and 7984 cells per second respectively, the latter having 348% increase over the former. More critically it is the difference between meeting the 2405 cells per second requirement or not.

Memory Requirements of Cost-Mapping Models

Alongside the execution time measurements discussed in Section 4.5.3 TFMin also produced precise RAM requirements for each model. TFMin stores all intermediate tensor buffers in a single pre-allocated memory space, therefore the memory requirement for a model is constant and known before it is compiled. Table 4.1 shows this RAM requirement alongside the weights storage requirement for each of the models proposed in Chapter 3.

It can be seen that the larger more efficient models require significantly more memory for both weights storage and to execute inference. The challenge presented by the weights of ML models has been a focus for the research community for some time. Model quantisation techniques are well established and benefit execution time, RAM requirement as well as the storage required for weights. Model pruning techniques identify and remove weights which do not contribute

meaningfully to output estimates, such as AutoML for Model Compression (AMC) proposed by Han et. al. [65]. Low-rank approximation is another family of techniques which attempt to compress a models weights by exploiting redundancy across dimensions, such as Jaderberg et.al. [79]. Techniques such as these as well as methods to reduce the amount of memory required by intermediate buffers will be especially beneficial to the larger and more efficient cost-mapping models, proposed.

Table 4.1: Memory requirements of proposed cost-mapping models.

Model	Weights Storage	RAM Required
Cnn-A	1120 KB	294 KB
Cnn-B	6350 KB	294 KB
Cnn-C	1631 KB	204 KB
Cnn-D	7910 KB	74 KB
Cnn-E	263 KB	93 KB
EncDec-A	9938 KB	428 KB
EncDec-B	13 MB	587 KB
EncDec-C	17 MB	979 KB
EncDec-D	3150 KB	428 KB
EncDec-E	5715 KB	587 KB
EncDec-F	9292 KB	979 KB
EncDec-X	24 MB	1462 KB

4.6 Summary

A new tool has been demonstrated which enables researchers and developers to easily manipulate and deploy ML models on a wide range of targets. Ranging from industry standard radiation hardened processors used on deep-space missions to low cost micro-controllers used in terrestrial IoT applications.

Initially being developed to automatically deploy ML models to the LEON3 processor when no other options were available, it now stands as one of three powerful open-source frameworks made available by Google [116], ARM [5], and Surrey Space Centre (SSC) & Airbus [23]. TFMin has been used to deploy ML models to the LEON3 & LEON4 radiation hardened

processors, ARM Cortex M series micro-controllers as well as desktop platforms. The ANSI C code generated is the language most widely supported by embedded CPU & DSP targets, since TFMin code requires no support libraries it generates the most portable implementations of all Edge ML frameworks.

Alongside this fundamental capability to deploy ML models TFMin provides an open extensible framework designed to facilitate experimentation into the science of ML inference on low power embedded devices. Powerful introspection functions are provided along with validation and verification tools supporting the development of graph mutation algorithms, memory pre-allocation algorithms, and other as yet unknown techniques. It has supported published research into the use of ML for terrain assessment on-board planetary rovers and novel memory optimisation techniques which could be applied to inference in any application domain.

This tool has been instrumental in facilitating my our research and is currently being used by Airbus to investigate deep-learning on-board future spacecraft. It is hoped that the open source release of this tool [23] will broaden its user base and impact within the ML research community. Alongside the open-source version of TFMin Airbus have ownership of the LEON3 & LEON4 features built during my research and are looking to commercially develop this product to support on-board ML development in the space sector.

Opportunities and Progress

New Research Opportunities Identified in this Chapter

- Development and verification of ML layer implementations optimised for the LEON family of radiation hardned CPUs.
- Investigate opportunities to reduce memory footprint of deployed models.
- Investigation into the effects of model binary size bloat on execution time.
- Investigate the effects of operation numerical stability on model accuracy.

Table 4.2: Opportunities at the end of the cost mapping investigation.

Challenge	Solution
Is it possible to generate planetary rover cost-maps using ML?	• Encoder decoder models trained using supervised learning have been shown to be effective.
Are ML cost-mapping models feasible to use on radiation hardened LEON3 computers?	• Execution time has been quantified on representative hardware, and found to meet the 20 second requirement from Airbus. • The RAM requirements of higher performing models are problematic for on-board software.
Does a deployment process exist to implement this model within the flight software development process?	• The TFMin tool has been developed and released which generates prototype ANSI C code suitable for the flight software development process.

Chapter 5

Memory Optimisation

5.1 Introduction

Limited processing power on smaller terrestrial CPUs is a hindrance when running ML models, but if longer execution time is acceptable then inference can still be performed. The amount of RAM however places a hard limit on the size of models which can be executed. If intermediate tensor buffers do not fit in the memory available on a target, there is no straightforward method to be able to execute the model. Meanwhile the memory constraints on radiation hardened computers used in space are also tight, on-board functionality must be maximised while maintaining strict resource margins.

Taking a small optimised model as an example (MobileNet v1.0 0.25 128 quantised to 8 bits [72]) when inference is being performed with this model the second 2D convolution operation requires 32 KB input and 64 KB output buffers. This operation defines the peak RAM requirement for this model at 96 KB as can be seen in the intermediate buffer allocation pattern shown in Figure 5.1. Figures of this type will be used several times in this chapter, they visualise intermediate tensor buffers in both memory location and scope, where location within memory is shown on the x-axis and scope (first use to final use) is shown on the y-axis. Both TFMin [22] and TFLμ [116] use a monolithic fixed size memory region known as the tensor arena to hold intermediate buffers. At the start of this work both TFMin and TFLμ used a heap allocation strategy to place tensor buffers in memory, TFMin performed this at compile time whereas TFLμ performed this at runtime.

Existing and proposed memory optimisation strategies for edge ML implementations are discussed in section 5.3 & 5.4, with particular reference to their novelty or similarity to techniques from other disciplines. Model compression techniques are described and their effectiveness at reducing RAM use discussed. It is shown that these techniques are complimentary, and indeed must be used together to achieve state-of-the-art reductions in RAM use. Existing work discussing the effect of different graph-sequencing algorithms on memory use is reviewed, and the potential memory savings analysed. In memory tensor representation approaches are discussed and their effect on the implementation of layout operations, this architectural decision is shown to significantly affect peak memory requirement.

Two novel techniques are presented that can reduce the amount of RAM required to perform inference, operation splitting, and DMO. Operation splitting is a graph level algorithm which breaks chains of large tensor operations into multiple parallel chains. These parallel chains can be executed in series using less RAM at the cost of a small increase in execution time. DMO works by determining a safe overlap between the input and output buffers of an operation, and uses this information to pre-allocate intermediate buffers more densely than current methods. Unlike operation splitting DMO has no execution time penalty.

The following sections discuss Operation Splitting and DMO in detail, including their theoretical basis and practical application. Experiments are conducted using a set of well known DL models comparing the memory requirements of the optimised models to existing state of the art implementations. It is shown that Operation Splitting can be used to reduce the peak memory requirement of MobileNet v2 by 73% while DMO can be used to reduce the peak memory requrement of MobileNet v1 by 33%. These two new memory optimisation techniques are applied to the twelve cost-mapping models proposed in Chapter 3, the memory savings and their impact is discussed.

The work described in this chapter has been released as a pre-print on the arXiv service:

- Blacker, P., Bridges, C.P. and Hadfield, S., 2020. Diagonal Memory Optimisation for Machine Learning on Micro-controllers. arXiv preprint arXiv:2010.01668.

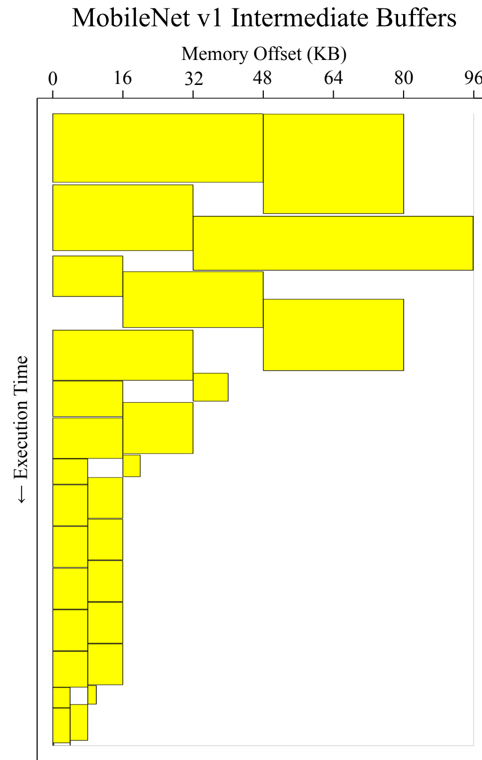


Figure 5.1: Intermediate tensor buffer locations for MobileNet v1 0.25 128, 8 bit quantised. Location within the tensor arena is shown on the x-axis while the scope of each buffer from first to last use is shown on the y-axis.

5.2 Problem Definition

The majority of research into the optimisation of ML inference has been focussed on speeding up execution and reducing model size, little work has been done solely to reduce the amount of RAM required for inference. The reason here is simple in almost all inference applications to date RAM requirements have not been a bottle-neck therefore using less memory for inference is of no benefit. However when performing inference using radiation hardened CPUs used on deep space missions, it is critical to optimise both time and memory use. Research in this area is also of benefit to the wider ML community which has recently started deploying models onto micro-controller scale targets. The common challenges faced when performing inference on terrestrial micro-controllers and radiation hardened processed increases the impact of this research which is motivated primarily by space applications.

Although model compression techniques are not primarily motivated by memory optimisation in many cases they achieve significant reductions in RAM use as a side effect. Their primary purposes are to reduce the amount of data needed to store or transmit models and to speed up inference. A common side-effect however is a reduction in the RAM required for inference. The clearest example of this is quantisation [164], which reduces the precision with which weights are stored and computations performed.

Training is most commonly performed using 32 bit floating point data, allowing for precise calculation of gradients for back propagation and allowing a wide range of input and training data to be used. Once a model has been trained the precision of the weights can be reduced with a small and acceptable reduction in accuracy. Weights as well as intermediate values are commonly reduced to 8 bits using predefined fixed point representation. This has the effect of dividing the model size by four as well as reducing the RAM requirement by four. Binary networks [75] take this concept to the extreme with a single bit representing each weight and intermediate value, for a further reduction in model size and memory use.

If the quantity of weights and intermediate values is constant reducing the size of the data type used to represent them has a predictable and linear effect on the amount of RAM required. The effect of weight pruning, compression, and low-rank approximation is more complex to define. Weight pruning attempts to ignore weights which do not contribute meaningfully to the output of the model, allowing a more compressed representation of the model. The effect of this technique on the RAM required for inference depends on the implementation used. If weights are removed in such a way that the tensor sizes used for fully-connected or convolution layers are reduced then a reduction in RAM will be realised. However the original tensor sizes are used with sparse representation and operations then any reduction in RAM requirement although possible, becomes difficult to predict.

Weight compression seeks to reduce the memory required to store a model using both lossy and lossless data compression methods [140]. This approach can actually increase RAM use if additional storage is needed to store weights after de-compression. Since the size and precision of the tensor calculation is not altered no direct reduction in RAM use is realised.

Low-rank approximation extends the concept of weight compression into the layer operation itself, attempting to simply a convolution calculation while also reducing the storage needed for

its weights [148]. This technique does not reduce the storage needed for intermediate values, although if weights are stored in RAM a reduction could be possible.

These model compression techniques share a common feature, they all alter the computation performed by the model in some way. Quantisation affects the precision and numerical stability of a model, while pruning, compression and low-rank approximation all subtly alter model weights. It is important observe that this is different to the memory optimisation algorithms presented in this chapter which do not alter the computation, down to the level numerical stability. It is therefore possible to use them in a similar manner to safe compiler optimisations, applying them automatically to models during deployment. Model compression techniques on the other hand require careful hand tuning and verification to ensure model accuracy does not degrade unacceptably. The two classes of technique are fully complimentary, the model compression techniques discussed can be applied side-by-side with either DMO or Operation-Splitting, to minimise the RAM required to perform inference.

5.2.1 Effects on Power and Latency

Research described in this chapter focusses on the amount of RAM which is required to perform inference using ML models. However the techniques presented have potential side effects on the energy use and latency of an implementation as well. A short discussion of these effects is presented here, although a more detailed study is warranted. This is especially true for EdgeML applications which are highly energy sensitive.

The radiation hardened computers used on deep space missions are not as energy optimised as the latest terrestrial EdgeML micro-controllers. In fact their memory has an additional overhead, periodic memory cleaning to fix radiation induced single bit errors using error correcting [25]. Energy use of memory is usually split into three contributors, background power, active power, and refresh power [32], cleaning of error correcting memory is an additional drain on radiation hardened computers.

Of the two memory optimisation techniques presented DMO has no impact on the number of load and store instructions issued, while operation-splitting causes a increase of less than 1 percent on all models tested. On radiation hardened computers they will have a minimal or no effect on the energy consumed during an inference operation. There is one possible exception to

this conclusion, the case when memory optimisation allows a smaller lower power RAM chip to use used. This is unlikely in all but the most rigorously optimised applications but when it occurs the improvement would be significant.

The effect of the DMO and Operation-Splitting techniques on energy use of terrestrial EdgeML micro-controllers is more complex to determine, given the more elaborate energy saving methods used in these devices. Restricting SDRAM refresh operations to memory addresses which are in use as opposed to whole banks would be a possibility, the complexity would be non-trivial although savings could be made. There remains valuable work to be done evaluating the memory efficiency of EdgeML implementations and identifying potential improvements.

5.3 Existing Approach to Reducing Peak Memory Use

Machine learning models are in essence graph functions comprised of tensor operations. On single core CPU targets these operations are executed sequentially to perform inference and sufficient RAM must be available to store the intermediate values needed during this process. Figure 5.1 shows the intermediate buffer locations and scopes for a MobileNet, allocated using a heap approach. In this case the peak memory requirement is defined by the third and forth buffers which are needed concurrently taking a total of 96 KB. Looking at these buffer allocations it does not immediately seem possible to reduce this memory requirement further, however upon deeper inspection methods to achieve this can be found.

5.3.1 Tensor Buffer Reuse

Tensor layout operations which re-arrange elements such as concatenate and pack are common in ML models and in some cases, Squeezenet [78], define a model's peak memory requirement. These operations do not compute new values, instead they re-project existing values into new shapes. If existing in-memory representations can be re-used then elements do not need to be copied or new buffers allocated. However this technique is only possible if sparse in-memory representation are supported. TFL only supports dense tensor representations meaning, for example, a split operation on any dimension above dimension zero will require new intermediate tensors to be allocated and elements copied into the new buffers, Section 4.3.5. TFMin on the

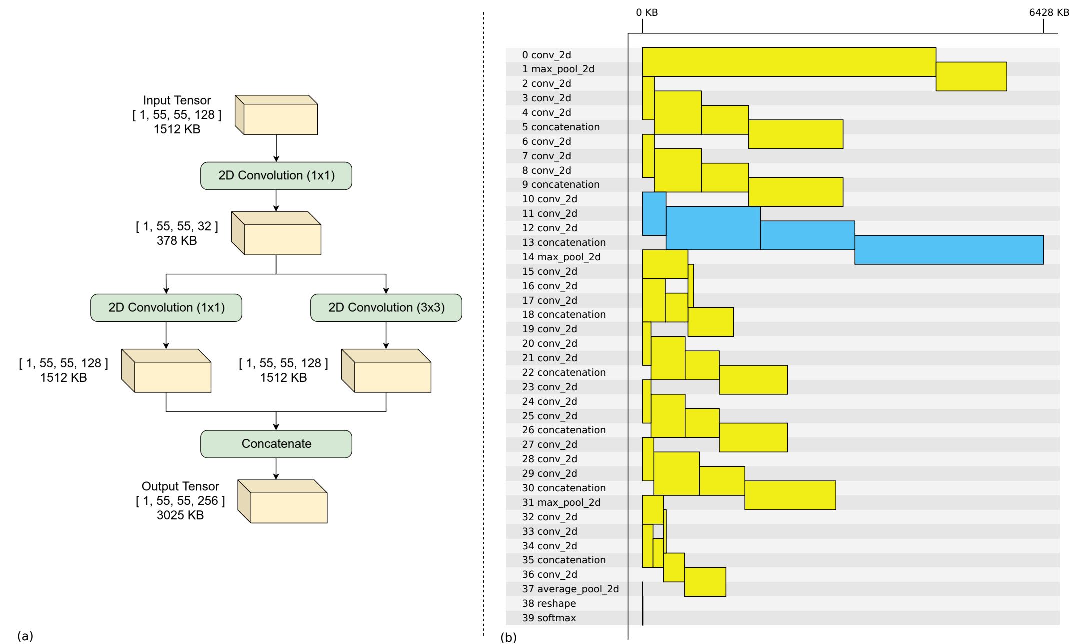


Figure 5.2: a, Sub-graph showing the 3rd fire module of the Squeezenet model. b, Intermediate buffer allocations of this model. It can be seen that the buffers of 3rd fire module (highlighted in blue) define the 6428 KB peak memory requirement of this model.

other hand supports sparse tensor representations so the output tensors of a split operation can be mapped onto the input buffer. If memory buffers are pre-allocated offline when a model is deployed this technique can also be used for concatenation operations. Instead of merging a set of smaller sub-tensors into a final large super-tensor, the operations which produce the sub-tensors write elements directly into the final merged tensor buffer. Taking advantage of sparse tensor representation in this way not only means that memory can be saved but also unnecessary data copying is avoided reducing execution time.

One of the reasons why TFL does not support sparse representations of tensors is the negative effect this can have on performance in some cases. If the innermost dimension is sparse (has a step greater than one element) then cache performance is negatively impacted and the use of vectorised load and store operations is no longer possible. However if higher dimensions are sparse there are no negative effects on performance since the offsets between slices is already greater than one element. These performance penalties coupled with the additional complexity of the tensor operations themselves meant that the architects of TFL chose dense representations. It is the authors view that in the context of aggressive memory optimisation then support for

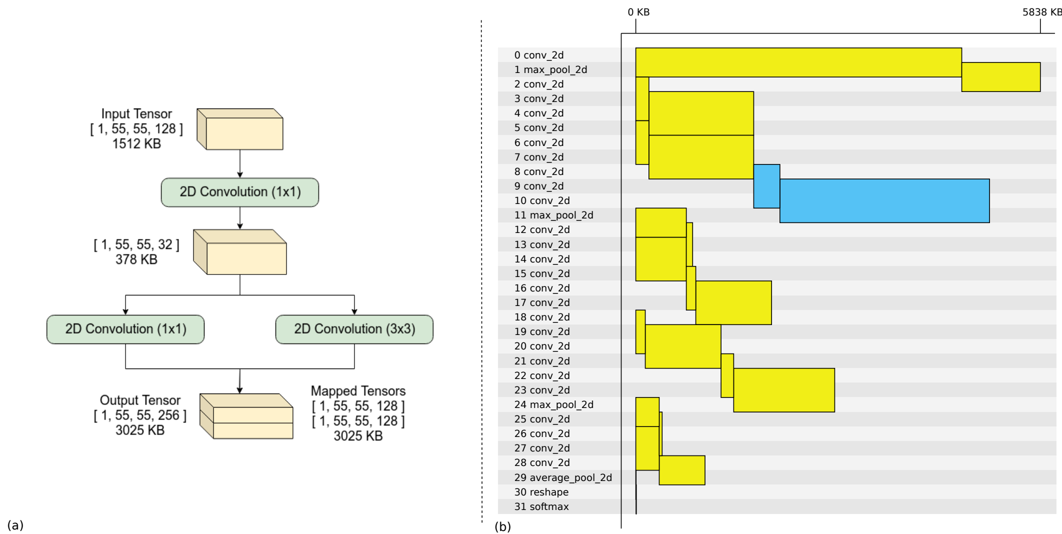


Figure 5.3: a, 3rd fire module of Squeezenet where the 2D convolution operations write directly into the super-tensor. b, Intermediate buffer allocations of this model. Buffers of the 3rd fire module are highlighted in blue. It can be seen that the optimised fire module no longer defines peak memory requirement of the model, it is now set by the the first max-pooling operation at 5838 KB.

sparse representations becomes preferable due to the significant saving it can produce in certain networks.

Using Squeezenet [78] as an example it can be seen in Figure 5.2.b that the 3rd fire module defines the peak memory requirement for inference using this model. The output tensors of the 2nd two convolution operations within this fire module can be mapped onto the output tensor of the concatenate, removing the need for the two of the intermediate tensors and the subsequent concatenate operation. It can be seen in Figure 5.3 that this modification to the fire modules reduces the peak memory requirement from 6428 KB to 5838 KB. The heap allocation strategy is producing a non-optimal layout of the fire modules buffers, however in this case since they are smaller than the first max pooling operation this does not affect the peak memory requirement.

The ability to remove layout operations via tensor mapping is dependant upon the architectural decision to support sparse in-memory tensors, taken when the inference system is specified. The decision to use this technique or not will always be a trade-off between the increased code complexity and runtime of models verses the value of any memory saving it produces.

Effects on Execution Time

As has been discussed, mapping tensors onto sparse in-memory representations can in some cases prevent optimised layer implementations being used. We argue that although it is theoretically possible, the cases are very specific and unlikely to occur in real-world models. To maximise performance TFL and TFMin both use a dimension ordering of batch, height, width, channels, to ensure that the inner loops of the majority of operations can work on adjacent values in memory. If sparse tensors are used but elements along the innermost dimension are kept in adjacent memory addresses then performance will not be affected.

In the Squeezenet example used, intermediate tensors are concatenated along the innermost dimension, so values written by the inner loop of the 2D convolution operation will occupy adjacent memory addresses even when written directly into the super-tensor. In this case there is no performance penalty for using sparse in-memory representation. The authors have performed the same check on all six of the network topologies evaluated in Section 5.5 and found that none of their layout operations could result in sparse inner dimensions if this type of optimisation is used.

5.4 Novel Techniques for Memory Optimisation

5.4.1 Operation Splitting

Chains of filter based operations (convolution, pooling, etc.) which require large intermediate buffers can be split into multiple narrower chains and executed sequentially. Modifying tensor graphs in this way means that fewer intermediate values need to be stored concurrently. Which can reduce the peak memory requirement of inference at the cost of some elements being computed more than once. This memory optimisation technique requires sparse in-memory tensor representation as described in Section 4.3.5, if only dense tensors are supported then the initial split and final merge operations will use more memory than is saved.

Demonstrating this technique using MobileNet v2 1.0 224, it can be seen that the second and third operations (a 2D convolution & depthwise convolution) between them process a 392 KB tensor into a 588 KB tensor. However the intermediate tensor between these two operations

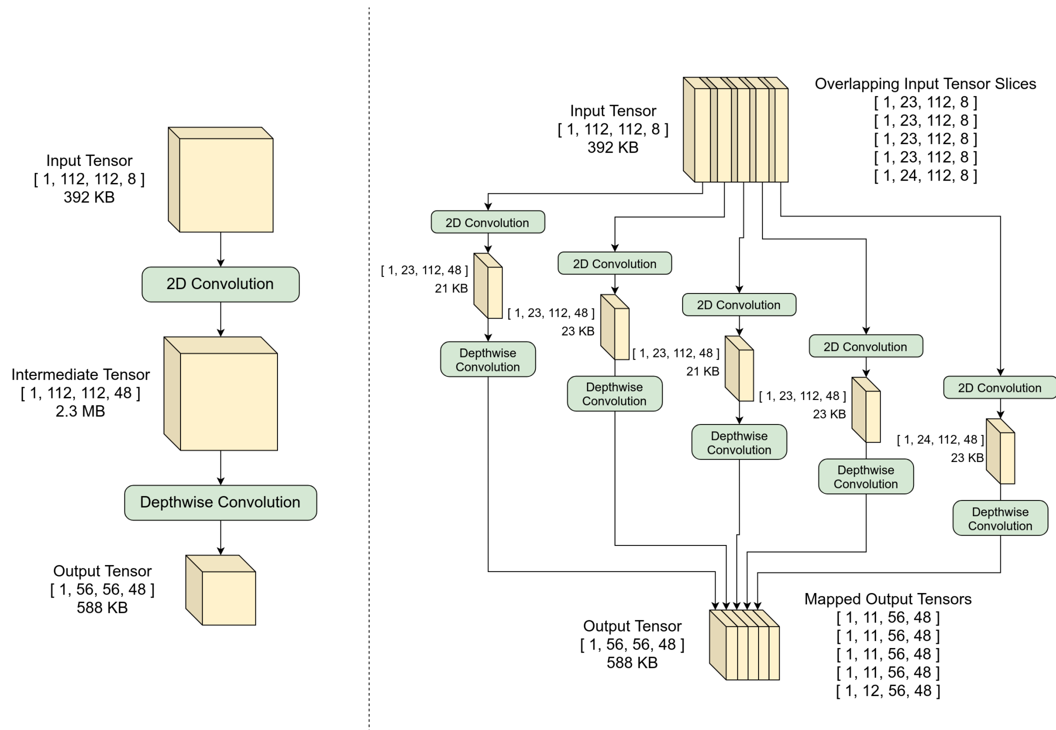


Figure 5.4: a, Subset of MobileNet showing the 2nd and 3rd operations before optimisation. b, Equivalent subset of the optimised graph, which computes the output tensor using five parallel pairs of operations. Note that there is a necessary overlap between the three intermediate tensors and input tensor slices because of the overlapping receptive fields of the depth-wise convolution operation.

takes 2.3 MB increasing the peak memory requirement of this model to 3.3 MB, Figure 5.5. Due to the small kernel sizes used in this model the receptive field of each element in the final 588 KB tensor is a $3 \times 3 \times depth$ patch of the 2.3 MB intermediate tensor. If the two operations are split into five pairs of operations, each pair computing approximately a fifth of the final output tensor then five consecutive intermediate tensors are needed of at most 819 KB Figure 5.4. This reduces the peak memory requirement of this part of the model to 1.6 MB, however due to the spatial overlap of the smaller intermediate tensors 10752 elements need to be computed twice. The allocation buffer locations before and after this alteration to the graph are shown in Figure 5.5. The longer scope of input and output tensors means that this approach can not be combined with conventional DMO described in Section 5.4.2 however the first input and final output tensors of the split operations can be potentially be overlapped.

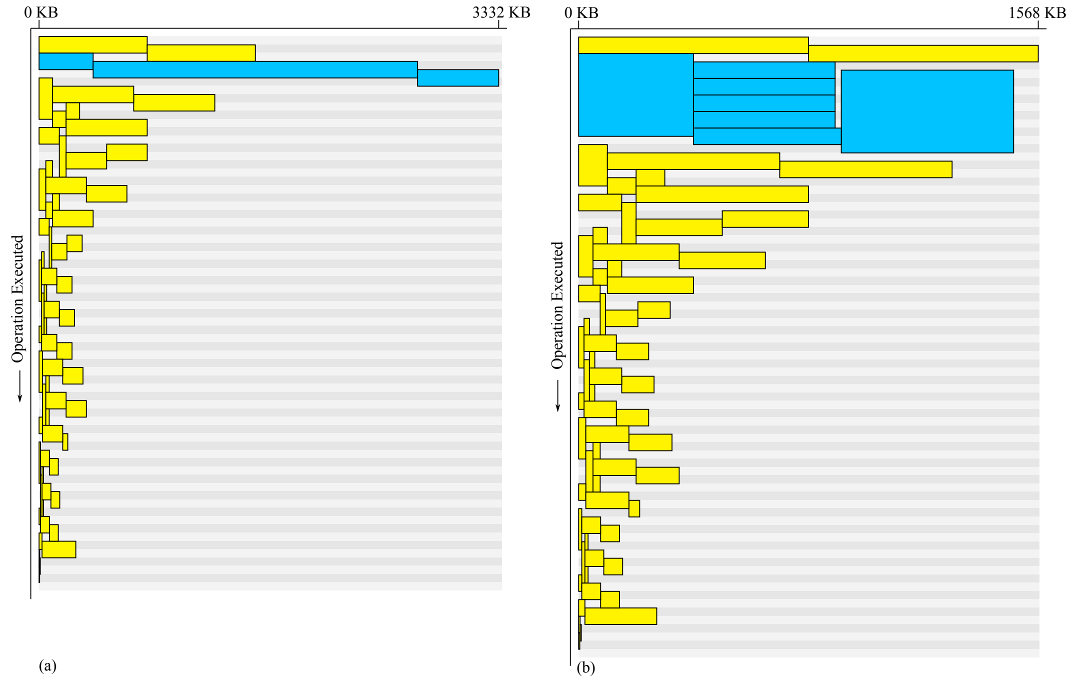


Figure 5.5: a, Intermediate buffer allocations of a full sized MobileNet V2 implementation. b, Intermediate buffer allocations of the same model in which the second and third operations have been split into five parallel branches. Buffers that have been effected by this optimisation are highlighted in blue.

The amount of memory saved and the number of element re-computations required when using this technique is formalised in Equations (5.1), (5.3) & (5.4). It can be seen that memory is saving is only possible if the intermediate tensor is larger than the input and output tensors, and the memory saving is greater for larger intermediate tensors. The test defined in boolean Equation (5.5) can be used to determine is operation splitting is possible for any pair of operations and if it yields any saving in memory.

$$Saving = size_{dtype}(in_s + out_s + int_{sn} - int_{so} - max(in_s, out_s)) \tag{5.1}$$

$$rec_s = (filter_{size} - 1)\, dilation_ratio + 1 \tag{5.2}$$

$$MaxInt_s(N) = \left(\frac{\prod(dims_m)}{splitdim_m}\right) max\left(2\left\lceil\frac{dim_o - 1}{N}\right\rceil, 2\left\lceil\frac{dim_o}{N}\right\rceil - R_{pad}\right) + rec_s - 2 \tag{5.3}$$

$$Recomputations = (N-1)\left(\frac{\prod(dims_m)}{splitdim_m}\right)(rec_s - stride) \quad (5.4)$$

$$SplitPossible = (op_1\ filter\ based) \land (op_2\ filter\ based) \land (IntSize(2) < int_{so}) \quad (5.5)$$

Where $size_{dtype}$ is the size of a tensor element in bytes, in_s is the size of the input tensor in bytes, out_s is the size of the output tensor in bytes, int_{so} is the size of the original intermediate tensor in bytes, $dims_m$ is the set of dimension sizes of the original intermediate tensor, $splitdim_m$ is the size of the split-dimension of the original intermediate tensor, rec_s is the size of the receptive field of the second operation in the split dimension, and N is the number of splits.

Using Equations (5.3) & (5.4) it is possible to build and algorithm that can analyse tensor graphs identifying potential memory savings and their cost in repeated element computations. Reducing peak memory requirements using operation splitting is a trade off between memory saving and execution time, therefore there is no single result of this optimisation technique. Rather a set of optimisation levels is determined including the memory saved and number of tensor elements which need to be recomputed. Developers can then choose which of these optimisation levels is most appropriate to their application. The algorithm described below produces this set of optimisation levels for any tensor graph.

Algorithm to Analyse Op-splitting Optimisation

This algorithm starts with the original input graph then iteratively attempts attempts to optimise it, initially the set of optimisation levels is the singleton set containing the non-optimised model. Each iteration which finds a more optimal model, adds an element to the set of optimisation levels and the algorithm continues iterating.

In order to determine if it is possible to optimise the model further the set of operations which define the peak memory requirement is found. In the initial MobileNet graph shown in Figure 5.5.a showing buffer allocations it can be seen that this operation is the 2nd 2D convolution. Whereas for the optimised graph who's buffer allocations are shown in Figure 5.5.b this operation is the 1st 2D convolution. It should be noted that in both these cases a single operation defines

Algorithm 1: Pseudo code of operation splitting analysis

$lastPeakMem \leftarrow pre_allocate_graph_get_peak_mem(initialGraph)$

$recomputations \leftarrow 0$

$setOfOptLevels \leftarrow [(initialGraph, lastPeakMem, 0)]$

$optimisedGraph \leftarrow initialGraph$

repeat

  $setOfPeakMemOps \leftarrow find_peak_mem_ops(optimisedGraph)$

  for $peakOp\ in\ setOfPeakMemOps$ **do**

    if Operation already split **then**

      increase number of splits by one

      $recomputations += increased\ recomputations\ from\ split$

    else

      $followingOp \leftarrow get_following_op(peakOp)$

      if $SplitPossible(peakOp,\ followingOp)$ [Eq (5.5)] **then**

        Split operation pair into two branches.

        $recomputations += additional\ recomputations$ [Eq (5.4)]

      end if

    end if

  end for

  $newPeakMem \leftarrow pre_allocate_graph_get_peak_mem(optimisedGraph)$

  $optimsed \leftarrow (newPeakMem < lastPeakMem)$

  $lastPeakMem \leftarrow newPeakMem$

  if $optimised$ **then**

    $setOfOptLevels.add((optimisedGraph, newPeakMem, recomputations))$

  end if

until $lastPeakMem = newPeakMem$

the peak memory requirement, although it is possible for multiple operations to define equal peak memory requirements in different parts of the graph.

Each operation which defines the peak memory requirement is checked, to find if splitting it and the following operation would reduce memory use. The reason for including the following operation is that the memory saving is caused by reducing the size of the intermediate tensor which passes data between these two operations, so both operations need to be split. There are two possibilities here, either these operations have already been split during a previous optimisation step, or they are original operations. In the first case where they have already been split then the algorithm increases the number of splits to further reduce the size of intermediate tensors. In the second case where the operations have not yet been split then the test defined in Equation (5.5) is used to determine if operation splitting is possible. If it is found to be possible then these operations are split into two parallel branches.

If it has been found that all operations in the set of operations defining the peak memory requirement can be split and that splitting them reduces the peak memory requirement then the savings and cost of this level is added to the set of optimisation levels and this optimisation process repeats again. The pseudo code of this algorithm is formalised in Algorithm 1.

Results of optimising Mobilenet v2 are shown in table 5.1, listing four possible optimisation levels. The first level produces a RAM reduction of 1.2 MB at a cost of recomputing 5376 elements of the 2nd convolution operation, while the most optimal level produces a reduction of 1.7 MB at the cost of recomputing 21504 elements. There are diminishing returns as the number of splits increases, due to the inverse relationship between the size of split tensors and the number of splits. To put these element re-computations into the context, the original network requires the computation of 2.94 million elements, so these optimisation levels represent increases of 0.18% and 0.73% respectively.

Operation-splitting can significantly reduce the peak memory required for inference of certain ML models, at the cost of a small increase in execution time. A formal analysis has been presented with an algorithm to determine potential memory savings and their costs. Allowing memory to be traded off against execution time, finding the optimal solution for a given application. The results of this technique on the full set of test models is shown in Section 5.5.

Table 5.1: Possible operation splitting optimisations found using algorithm.

Optimisation Level	Peak Memory	Memory Saved	Elements Recomputed
original graph	3332 KB	0 KB	0
[2:conv, 3:dw_conv split 2]	2177 KB	1155 KB	5376
[2:conv, 3:dw_conv split 3]	1799 KB	1533 KB	10752
[2:conv, 3:dw_conv split 4]	1589 KB	1743 KB	17498
[2:conv, 3:dw_conv split 5]	1568 KB	1764 KB	21504

5.4.2 Diagonal Memory Optimisation

The opportunity to reduce memory use that is leveraged by DMO was discovered while analysing the detailed memory use patterns of implemented ML models. A customised version of the Valgrind debugging tool [111] was developed which can observe memory read and write operations within the tensor arena while a model is being executed. This analysis determines areas of memory which are holding values which are later read and used for calculations, and redundant areas of memory where the value is never read. Figure 5.6.a shows a memory trace produced by this tool of the same MobileNet implementation described in Figure 5.1. Exposing the internal workings of each operation in this way an opportunity can be clearly seen to reduce the peak memory requirement by overlapping the input and output buffers of each operation.

DMO can significantly reduce the peak memory demands of machine learning models but is only possible if the underlying layer implementations are known and methods exist to determine the safe overlap offset of a specific operation instance. This requires an architectural change from pre-allocation schemes currently used by TFLμ and uTensor from ARM [5], where intermediate buffers are allocated without any knowledge of the layer implementations which will eventually use them. The task of computing the safe buffer overlap (O_s) for any operation implementation is investigated in Section 5.4.3 and three different methods presented. Diagonal memory optimisation with performance optimised layer implementations is the discussed with reference to vectorisation and multi-threading.

Calculating the safe input/output buffer overlap for tensor operations is not a task that has been

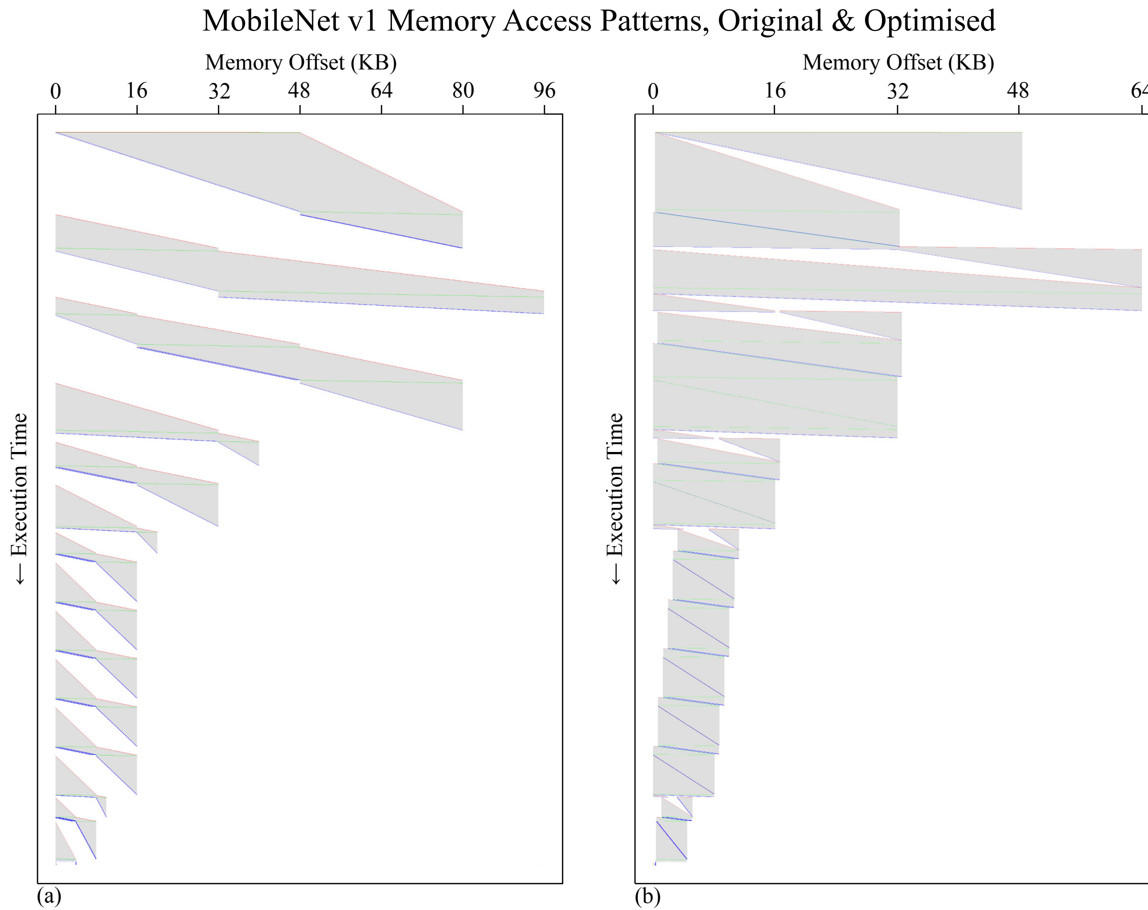


Figure 5.6: Intermediate buffer memory access pattern for the example model (MobileNet v1 0.25 128 quantised). In use areas shown in grey, load, store, and update events in red, blue, and green respectively. Plot a shows the memory access pattern when the original heap allocation strategy is used to allocate intermediate buffers, large areas of unused memory can be see which could be used to reduce the size of the tensor arena. Plot b shows the memory access pattern of the same model with intermediate buffers allocated using diagonal memory optimisation, in-use memory is packed more densely allowing the size of the tensor area to be reduced.

described in literature before. Yet it is an essential capability of tools that can automatically leverage this optimisation strategy. This work investigates three approaches to the calculation of O_s and presents a set of analytical solutions for common tensor operations. These solutions are evaluated by generating buffer pre-allocation patterns for eleven notable ML models, memory reductions of up-to 49% were achieved. Figure 5.6 illustrates how DMO reduces the amount of memory MobileNet v1 requires by packing in-use areas of memory together more densely than

is possible using block level optimisers.

Models implemented using optimised buffer allocation patterns generated by DMO have been shown to be mathematically correct using the verification functions of TFMin described in Section 4.4.3. Original and optimised graphs were executed side-by-side and their outputs shown to be identical.

The DMO technique proposed here works at a lower level than the graph based approaches described in sections 5.3 & 5.4.1 and is complimentary to the graph serialization and operation removal techniques. The observation that the input and output buffers of many tensor operation can be safely overlapped is utilised in cases where the input to an operation is not needed by any later operations. Therefore the input buffer can be safely overwritten during the computation of the operation's output.

A heap based allocation approach was used to place intermediate tensor into the tensor arena in reverse execution order. Reverse ordering was used because diagonal memory optimisation allows the start of the input buffer to overlap with the end of the output buffer, therefore allocating the input buffer after the output buffer means the heap strategy can overlap the buffers efficiently in most cases. The result of the DMO approach with buffer overlapping can be seen in Figure 5.6.

5.4.3 Calculating the Safe Buffer Overlap

To understand the definition of the safe buffer overlap three methods to determine O_s are presented. Our initial work debugging compiled networks is described first, followed by a more efficient algorithmic approach. Ultimately a method to derive analytical lower bounds of O_s is presented along with a discussion of their precision. Finally the effect of performance optimisation techniques on the utility of buffer overlapping is discussed.

Definition of the Safe Buffer Overlap

It can be intuitively seen in Figure 5.7 that the input and output buffers of three of the four tensor operations can be overlapped a certain amount without any values in memory being clobbered.

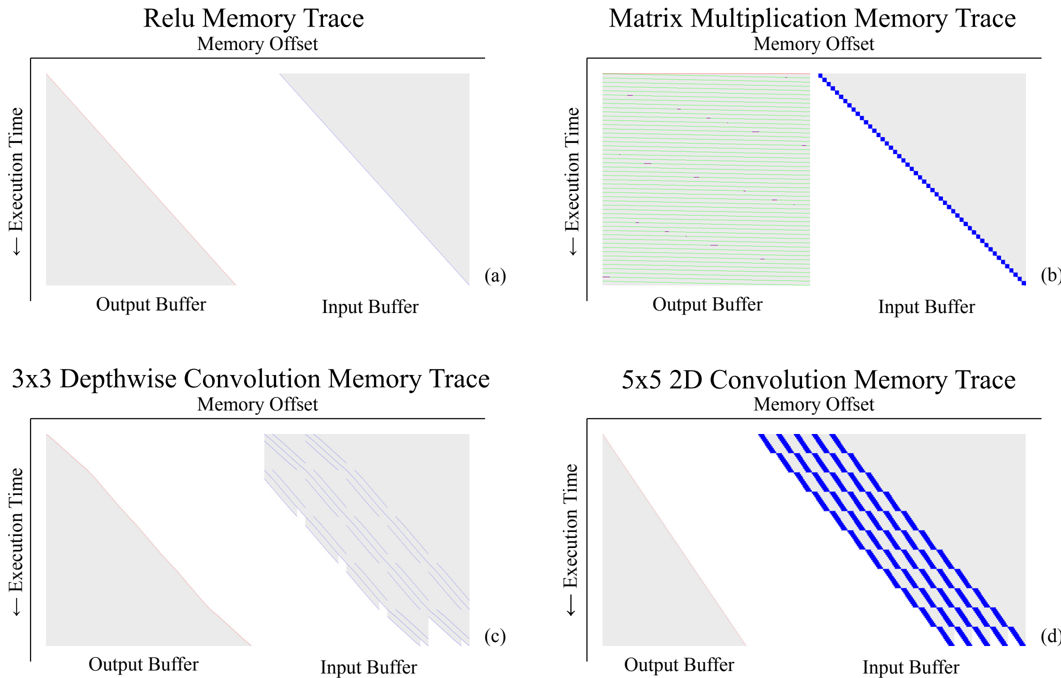


Figure 5.7: Memory traces of four common ML tensor operations. (a) Rectified Linear Unit, (b) Matrix Multiplication, (c) Depthwise Convolution, (d) 2D Convolution. These traces only show intermediate input & output tensor buffers, ignoring the filter and weight buffers.

The type of operation, algorithm used to compute the result, size inputs, and parameters all determine its pattern of memory access and therefore the exact size of this safe overlap.

Element-wise unary and binary operations such as the Relu shown in Figure 5.7 a, have perfectly diagonal input and output patterns representing the ideal case where O_s is the size of the output buffer. It is interesting to note here, that in-place buffer re-use is actually a special case of diagonal memory optimisation. The matrix multiplication operation shown in Figure 5.7 b represents the other extreme, the whole range of its output buffer is repeatedly updated until the final slice is processed. In this case the input and output buffers can not be overlapped at all. Depth-wise convolution and 2D convolution operations shown in Figures 5.7 c & d fall somewhere between these two extremes.

By convention algorithms progress from lower indices to higher indices, for simplicity this work assumes that algorithms will always be processed in this direction. Although it is theoretically possible to make use of algorithms which can process in either direction, this work has not investigated this option. Safe buffer overlap O_s is formally defined as the maximum number

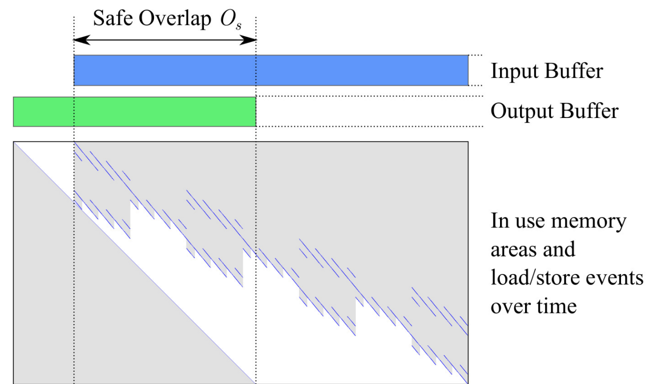


Figure 5.8: Definition of the safe buffer overlap (O_s) metric, defined as the maximum overlap where no in-use areas of memory are clobbered. In use memory shown in grey, write operations in red, and read operations in blue.

of bytes that the start of the input buffer can be overlapped with the end of the output buffer without clobbering any values in memory, as shown in Figure 5.8. The memory saved for each operation is equal to the buffer overlap O_s itself. The process of determining O_s and using the safe overlap when allocating intermediate buffers is the central new concept of diagonal memory optimisation.

Bottom up Method

Bottom up approaches such as the Valgrind [111] method described in Section 5.4.2 observe the load and store operations of a compiled operation while it is executed. The difference between this and a conventional memory trace is a mechanism to isolate the memory operations of the layer implementation from the rest of the compiled binary. The authors developed a tool (VMT Section 4.4.2) which in combination with Valgrind is able to record memory traces of specific memory ranges within a test binary. This tool was used to record the memory access patterns of single layer operations and whole ML models, identifying the original opportunity for memory optimisation of MobileNet shown in Figure 5.6.

The raw output of this introspection method is a set of memory events at 2D locations in time and buffer-offset, measured in instructions and bytes respectively. This type of raw output was used to produce the plots shown in Figure 5.7 showing the memory access patterns of four types

of operation. It is important to note that memory offset of these events is not recorded as a raw memory address but as a byte offset from the known start of the tensor buffer. Because these events contain buffer offsets they can be processed to find the maximum safe overlap O_s between the input and output buffers of the operation being analysed.

To calculate O_s first all the memory events need to be in a single list sorted in ascending order of instruction. Then two corresponding lists ($minRead$ & $maxWrite$) with one-to-one mapping to the event list are populated. The $minRead$ list contains the lowest read address of this and all following events and the $maxWrite$ list contains the highest write address of this and all preceding events. The values of both these lists can be populated with forwards and backwards pass. The value of O_s is then be computed using Equation (5.6).

$$O_s = output_s + \min_{i \in I} (minRead[i] - maxWrite[i]) \tag{5.6}$$

Where $output_s$ is the size of the output buffer in bytes and I is the set of indices to the $minRead$ and $maxWrite$ lists.

The advantage of the bottom-up method is that layer implementations can be a black-boxes, even implementations in compiled libraries can be analysed meaning this approach can be used with many existing operations. There is however a requirement that memory read/write behaviour must be deterministic, which excludes multi-threaded implementations due to the non-deterministic nature of thread synchronisation. This limitation applies to diagonal memory optimisation itself which is discussed further in Section 5.4.3. A drawback of this method is its high computational cost and complexity of the process. Building dedicated test binaries and debugging layer implementations is a complex approach, the algorithmic and analytic methods described offer faster and more portable approaches to find O_s but require access to original source code.

Algorithmic Method

The algorithmic method requires the development of a new algorithm to compute O_s based upon the original layer implementation. This new algorithm removes the calculation of tensor

values leaving only the calculation of buffer offsets where these values would have been read from and written to.

The original implementation is analysed so that the number of write or update operations ($Steps$) on the output buffer can be determined. A new algorithm is then written which produces two arrays $minR$ & $maxW$ each $Steps$ long. Where each element of $minR$ contains the minimum read offset of that step and all future steps, while each element of $maxW$ contains the maximum write offset of this step and all previous steps. O_s can then be calculated using Equation 5.7 where O_b is the output buffer size and $minD$ is the minimum of $minR - maxW$ across the arrays. This conceptual structure of the algorithmic method defines the requirements for it to be applied to any deterministic tensor operation.

and is enough for a developer to write a new algorithm to compute for O_s for any existing deterministic algorithm.

$$O_s = O_b + minD \tag{5.7}$$

A practical demonstration of this method is presented using pseudo code of the depthwise 2D convolution reference implementation Appendix B.1 from TFL shown in Algorithm 2. Bias and activation functions have been omitted for clarity since they have no effect on the computation of O_s.

In this case a single output element is computed within each iteration of the 5th nested loop, therefore the $minR$ & $maxW$ arrays need to be $batches \times outputH \times outputW \times inputD \times filterC$ elements long. The value of $minR$ can be found for each iteration as the minimum of all values computed within the $filterY$ & $filterX$ loops, as long as a final reverse pass is performed to enforce the 'minimum of all future iterations' requirement. The value of $maxW$ for an iteration is the highest value of O_o computed so far through the loops. These modifications are shown in Algorithm 3.

An implementation of the pseudo code above can be used to calculate the value of O_s for any instance of the reference depthwise 2D convolution operation directly without the need to inspect the behaviour of a compiled layer. The pattern of code changes in the demonstration above can be applied to any single-threaded tensor operation, converting it into an algorithm

Algorithm 2: Depthwise 2D Convolution - Pseudo Code

```
for b = 0 to batches do
    for outY = 0 to outputH do
        for outX = 0 to outputW do
            for ic = 0 to inputD do
                for m = 0 to filterC do
                    total ← 0 for filterY = 0 to filterH do
                        for filterX = 0 to filterW do
                            if input element in input tensor then
                                F_o ← [calc filter offset] I_o ← [calc input offset]
                                total ← total + (filter[F_o] × input[I_o])
                            end
                        end
                    end
                    O_o ← [calc output offset] output[O_o] ← total
                end
            end
        end
    end
end
```

Algorithm 3: Computation of O_s - Pseudo Code

$Steps \leftarrow batches \cdot outputH \cdot outputW \cdot inputD \cdot filterC$ $minR = array(iCount)$
$maxW = array(iCount)$ $maxF_o = 0$ $it = 0$ **for** $b = 0$ *to* $batches$ **do**
  for $outY = 0$ *to* $outputH$ **do**
    for $outX = 0$ *to* $outputW$ **do**
      for $ic = 0$ *to* $inputD$ **do**
        for $m = 0$ *to* $filterC$ **do**
          $minR_o \leftarrow +inf$ **for** $filterY = 0$ *to* $filterH$ **do**
            for $filterX = 0$ *to* $filterW$ **do**
              if *input element in input tensor* **then**
                $I_o \leftarrow$ [calc input offset]
                $minR_o \leftarrow min(minR_o, I_o)$
              end
            end
          end
          $minR[it] \leftarrow minR_o$ $O_o \leftarrow$ [calc output offset]
          $maxW[it] \leftarrow max(maxF_o, O_o)$ $it \leftarrow it + 1$
        end
      end
    end
  end
end
$minR_o \leftarrow +inf$ $minD \leftarrow 0$ **for** $i = Steps$ *to* 0 **do**
  $minR[i] \leftarrow min(minR[i], minR_o)$
  $minD \leftarrow min((minR[i] - maxW[i]), minD)$
end
$O_s = outputBufSize + minD$

for the direct computation of O_s. In this specific example further inspection of the source code reveals that the values of $minR_o$ and O_o calculated by the first set of loops will always be monotonic with respect to it. Therefore in this case the code could be simplified to a single set nested of loops.

The algorithmic method is faster and more convenient than the bottom-up method, however it still requires a set of large nested loops to be executed. Since O_s is generated by complex algorithms it is difficult to generalise the solutions between different types of layer implementation. These shortcomings are addressed using by analytical method described in the following section.

Analyical Method

Using the Analytic approach an equation is derived for a specific layer implementation which directly calculates O_s for any instance of that layer. This approach requires the least computation time and more importantly is the least error-prone when translating between programming languages. We describe the approach taken to derive these analytical solutions for several common ML layer implementations. These equations were then used to generate optimised buffer pre-allocations of the eleven test models shown in Section 5.5.

It is important to note that useful solutions for the safe buffer overlap function, do not need to be exact, lower bound estimators will not break the operation while still reducing memory use. Meaning that analytical solutions can simplify certain details for convenience and still be of use. Similarly to the algorithmic method the analytical method requires a developer to analyse each new operation and to derive suitable equations, so that a memory optimisation algorithm can then allocate buffers both efficiently and safely.

The first stage of this analysis is to distil the memory access behaviour of an operation into two functions $minR(i)$ and $maxW(i)$ where i is equivalent to $Steps$ as defined in the algorithmic method. These two function $minR(i)$ and $maxW(i)$ have the same meaning as the arrays defined in the algorithmic method, this approach however derives equations for them. Figure 5.10 shows an example of the derived monotonic $minR(i)$ function for depthwise 2D convolution, it can be seen than all read operations, shown in blue, are bounded by the function shown in green.

Using these two functions O_s can then be found using Equation (5.8) where i_c is the total number of iterations, OB_s is the size of the output buffer, and T_s is the tensor element size

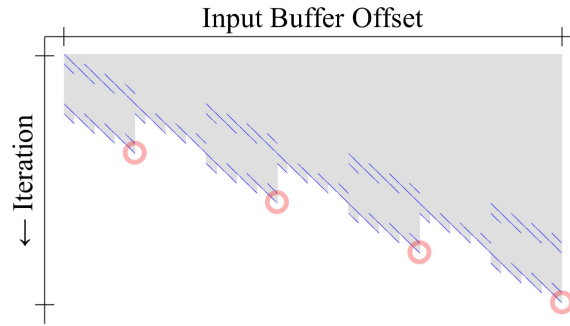


Figure 5.9: Memory read pattern example from a depthwise 2D convolution. Points highlighted define a linear boundary containing all read operations.

in bytes. Strictly this is enough information for a developer to be able to derive an analytic solution for O_s for any algorithm. However for clarity the derivation of the O_s equation for a reference depthwise 2D convolution is shown to illustrate how this can be applied to typical layer implementations used in ML models.

$$O_s = OB_s + min\{minR(i) - maxW(i) : i \in \mathbb{Z} \wedge 0 \geqslant i \geqslant i_c\}T_s \tag{5.8}$$

The approach to solving O_s here is the similar to the algorithmic method, except given analytic solutions for both $minR(i)$ and $maxW(i)$ then Equation 5.8 can be simplified to an analytic solution. Using these analytic solutions O_s can be computed directly without needing to loop through a large simulated tensor operation, potentially taking millions of iterations.

Taking the algorithmic solution for O_s of the reference DepthwiseConv2D operation used above we can study it and derive a purely analytical solution for the lower bound of $minR(i)$. Figure 5.9 shows the pattern of memory reads for an instance of this layer operation, the reads highlighted with red circles define a linear function bounding all reads of this algorithm. Firstly the location of the highlighted operations within the loops of the algorithm must be determined. Secondly equations for the iteration and read offset locations of these points must be derived.

By studying the pseudo code in Algorithm 2 it can be found that the exact read operations for this operation highlighted in Figure 5.9 occur during every iteration of the $outY$ loop and the final combined iteration of the $outX$, ic and m loops. Knowing this we determine that each of the highlighted reads occurs where $outY = N, outX = outputW - 1, ic = inputD - 1, m =$

$filterC - 1$. Next we determine that the minimum read within this iteration will always occur when $filterY = 0, filterX = 0$. Using this information, pseudo code, and the helper functions shown in the original C++ code [Appendix B.1] two Equations (5.9) & (5.10) for the values of the output offset o and iteration I in terms of N and the layer parameters can be found. These equations are derived by merging the code of the layer function itself with the helper functions $Offset$ and $ComputePadding$ and simplifying.

$$i = (N \cdot O_w O_d K_c) - 1 \tag{5.9}$$

$$o = Offset(N \cdot S_w - P_w, (O_w - 1)S_h - P_h, I_d - 1) \tag{5.10}$$

Where:

$$Offset(r, c, d) = (r \cdot I_w + c)I_d + d \tag{5.11}$$

$$P_h = \left\lfloor \frac{O_h S_h - S_h + K_h D_h - D_h - I_h + 1}{2} \right\rfloor \tag{5.12}$$

$$P_w = \left\lfloor \frac{O_w S_w - S_w + K_w D_w - D_w - I_w + 1}{2} \right\rfloor \tag{5.13}$$

Where I_w & I_h are input shape, O_w & O_h are output shape, K_w, K_h & K_c are kernel size, S_w & S_h are stride steps, D_w & D_h are dilation ratios.

The equations for these points (5.9) & (5.10) can then be used to define a linear function ($ax+b$) which bounds all read operations of this layer implementation. Simplifying these gives the Equations (5.14) & (5.15) for a & b respectively.

$$a = \frac{S_h I_w}{O_w K_c} \tag{5.14}$$

$$b = (O_w S_w - P_h I_w - S_h I_w - S_w - P_w + 1)I_d \tag{5.15}$$

Truncating this linear function at zero gives Equation 5.16 defining a lower bound approximation of the ideal $minR(i)$ as shown in Figure 5.10.

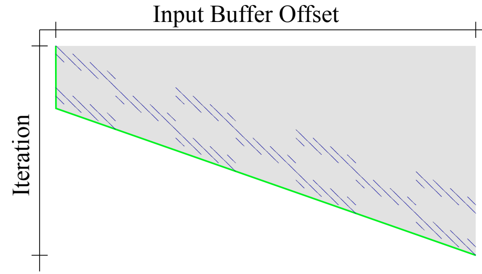


Figure 5.10: $minR(i)$ bounding function for the depthwise 2D convolution implementation. It can be seen that all read operations (in blue) lie above the monotonic function (green).

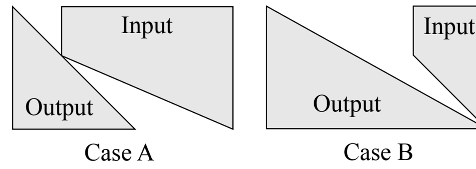


Figure 5.11: The two possible definitions of the analytical minimum bound, depending on the relative gradient of the $minR$ & $maxW$ functions.

$$minR(i) = max(0, a \cdot i + b) \tag{5.16}$$

The function $maxW(i)$ is trivial in the case of this operation, since each iteration calculates a single element of the output tensor and the loops are nested in increasing dimension order, therefore.

$$maxW(i) = i \tag{5.17}$$

Equations (5.8), (5.16) & (5.17) can be combined into a single Equation (5.18) by observing that O_s is defined by the $minR$ & $maxW$ functions in only two possible ways, as shown in Figure 5.11. If the gradient of $maxW$ is lower than that of $minR$ then it is defined as in case A otherwise it is defined as in case B. These two cases result in the two terms of the min function in the simplified analytical solution for O_s shown in Equation (5.18). This part of the analytical solution can be used to describe O_s for a wide range of tensor operations, including 2D convolution, all pooling operations, the depthwise 3D convolution described here and more. The only variation of this form are the equations for a and b which define the truncated linear bound of their read offsets.

$$O_s = OB_s + min\left\{\frac{b}{a}, ai_c + b - i_c\right\}T_s \tag{5.18}$$

Using this same process the minimum bound linear functions of several common ML reference operations have been derived. Combining these with Equation (5.18) produces their respective analytical solutions for O_s.

Reference 2D Convolution Operation:

$$a = \frac{S_h I_w I_d}{O_w O_d} \tag{5.19}$$

$$b = (O_w S_w - P_h I_w - S_h I_w - S_w - P_w)I_d + 1 \tag{5.20}$$

Reference Pooling Operations (all types):

$$a = \frac{S_h I_w}{O_w} \tag{5.21}$$

$$b = (O_w S_w - P_h I_w - S_h I_w - S_w - P_w)I_d + 1 \tag{5.22}$$

Precision of the Analyical Method

Since the analytical solutions presented in Section 5.4.3 are lower bounds as opposed to the exact values computed by the algorithmic method in Section 5.4.3, it is important to quantify the difference between the two. Using MobileNet v2 1.0 224 as an example when DMO is used, its peak memory requirement is defined by the second depthwise 2D convolution operation described in Table 5.2.

Computing the O_s of this operation using the algorithmic method gives a result of 1,204,224 bytes, while computing it using the analytic solution presented in Equations (5.14) (5.15) & (5.18) gives a result of 1,193,376 bytes. In this case the exact value has been underestimated by 10,848 bytes or 0.18% of the models optimised memory requirement of 4.6 MB, this is considered to be an acceptable approximation. Table 5.3 shows this same measurement for

Table 5.2: Specification of 2nd Depthwise 2D Convolution in MobileNet

Setting	**Value**
input shape (w, h, c)	112, 112, 96
filter shape (w, h, in_c, out_c)	3, 3, 96, 1
output shape (w, h, c)	56, 56, 96
stride (w, h)	2, 2
dilation (w, h)	1, 1

Table 5.3: Estimation Error of Safe Overlap (O_s)

Model	**Safe Offset** (O_s)		**Error**
	Exact	Estimate	
mobilenet v1 1.0 224	1204224	1193376	0.18%
mobilenet v2 1.0 224	1605632	1598400	0.15%
Inception ResNet v2	2746884	2746884	0%

three networks, comparing the exact algorithmic result with the lower bound from the analytical method. The underestimation of O_s ranges from 0% to 0.18%.

Performance Optimised Layer Implementations

The reference depthwise 2D convolution used to illustrate the computation of O_s is not the most computationally efficient implementation. This implementation is commonly used for smaller models on embedded ML applications however more efficient versions are increasingly being used that are optimised for specific processor families, such as cmsis-nn from ARM [88]. It is important to determine if safe buffer overlapping is possible when using these faster implementations, and to discover if the process described above to derive the analytical solution to O_s is still valid.

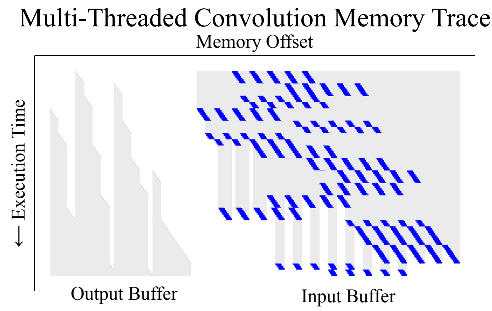


Figure 5.12: Memory trace of a 5×5 2D Convolution operation being executed using four threads.

Two optimisation approaches commonly used in embedded ML applications are vectorisation and multi-threading. Vectorisation takes advantage of Single Instruction Multiple Data (SIMD) operations to accelerate batches of identical arithmetic operations and reduces loop overheads. While multi-threading can be used when computations as opposed to memory access are the performance bottle-neck. These two approaches are not mutually exclusive and are often used together.

Vectorisation compliments diagonal memory optimisation because multiple elements are processed in longer words, therefore some reads occur earlier and some writes occur later. However it is always possible that trailing elements may need to be computed individually, for this reason the value of O_s for vectorised optimisation is the same as for the reference operations already described.

Multi-threading as it is usually implemented reduces the utility of using diagonal memory optimisation. Threads are generally each given a different contiguous region of the output buffer to compute, resulting in a memory access pattern similar as shown in Figure 5.12. It must be noted that the Valgrind tool used to generate this trace interleaves threads on a single core so does not precisely reproduce true multi-threaded behaviour. However two important features can be seen, firstly four different regions of the output buffer are being computed at similar times and secondly the read/write pattern has become non-deterministic. It is possible to overcome these problems by interleaving the elements each thread computes and ensuring that threads are synchronised to within a maximum offset. It is therefore theoretically possible to write buffer-overlap safe multi-threaded layer implementations, however more work is needed to demonstrate the practicality and reliability of the technique in this case.

5.5 Results

Eleven well known, published ML models were analysed along with the twelve cost-map estimation models proposed in Chapter 3. The published models were chosen a with particular emphasis on smaller models which have been developed for mobile applications. Models were sourced from the public repositories Keras Application [137] and TensorFlow models [8]. Eighteen purely sequential models were analysed, the variants of both MobileNet v1 and v2 as well as all of our proposed cost-mapping models. The sequential nature of these models makes them more amenable to optimisation than the more densely connected networks which have been analysed. Inception v4, Inception ResNet, Nasnet Mobile, DenseNet and ResNet 50 are all densely connected networks, as will be shown this makes the savings of DMO and operation-splitting harder to predict than for sequential models.

A modified heap allocation algorithm was used to locate the tensors buffers for each test and thereby find the peak memory requirement for each combination of model and optimisation strategy. This algorithm uses conventional heap allocation to to place tensor buffers in memory, but chooses a specific order in which to allocate buffers which has been found to reduce the peak memory requirement. The next buffer to allocate is chosen using two steps. First the set of un-allocated tensors which have scopes overlapping with allocated buffers is found. Out of this set the buffer is chosen which can be packed into the lowest address space. This algorithm is initiated by allocating a single input or output buffer at offset zero, to perform a forwards or backwards allocation respectively.

The Operation splitting method produces a set of optimised graphs with a range of performance penalties. The results shown here represented the optimised graph with the smallest peak memory requirement for each model analysed. Element re-computations are also shown to put the performance penalty of this method in context.

5.5.1 Sequential Published Models

DMO produces nearly identical memory savings for all of the MobileNet v1 variants analysed in this work. In the case of these models the original peak memory requirement is defined by the second convolution operation which produces an output tensor two times larger than

its input tensor. DMO allows these input and output tensors to overlap almost completely, reducing the peak memory requirement by a third. The small variation in the exact saving is caused by the amount the buffers cannot be completely overlapped due to the kernel size of the convolution. The ratio of this kernel size and the tensor buffer size varies depending on the size of the MobileNet variant in question, explaining this small difference. The pattern of tensor buffer allocations with and without DMO for MobileNet v1 can be seen in Figure 5.6.

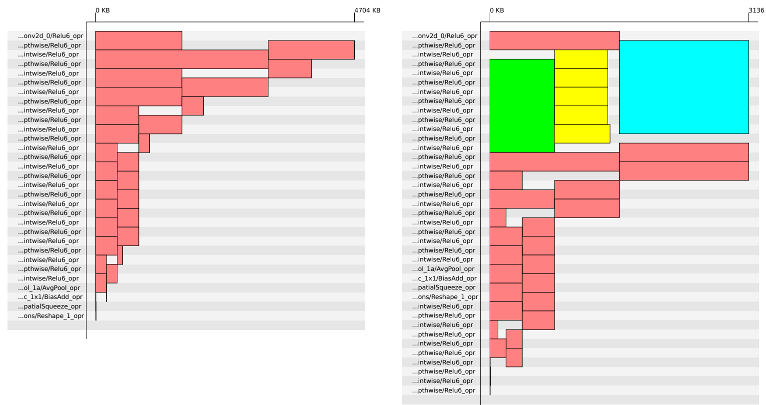


Figure 5.13: a, Original buffer allocation pattern of MobileNet v1 2.0 224. b, Optimised buffer allocation pattern after operation-splitting has been applied to the graph.

DMO when applied to the variants of MobileNet v2 produces a lower memory saving of 20%. In the case of these networks the peak memory requirement of the original graph is defined by the second depth-wise convolution operation which has an input tensor four times larger than its output tensor. Again DMO can almost completely overlap these two buffers, but because there is a greater disparity in their sizes this results in a lower memory saving than the MobileNet v1 variants. This allows us to make an important observation, that the memory saving of DMO will be greatest if the peak memory of a model is defined by two equal sized tensors. As the size disparity of the two tensors increases then the memory saved by DMO will reduce. Close to ideal DMO savings of up-to 49.2% were observed in the encoder-decoder cost-mapping models shown in Table 5.6.

When operation-splitting is used to optimise the memory use of the MobileNet v1 variants analysed we again see a memory saving of 33% in most cases, although the reasons behind this figure are different to those for DMO. The peak memory requirement of the original model is defined by the second convolution operation, operation-splitting splits this operation and its

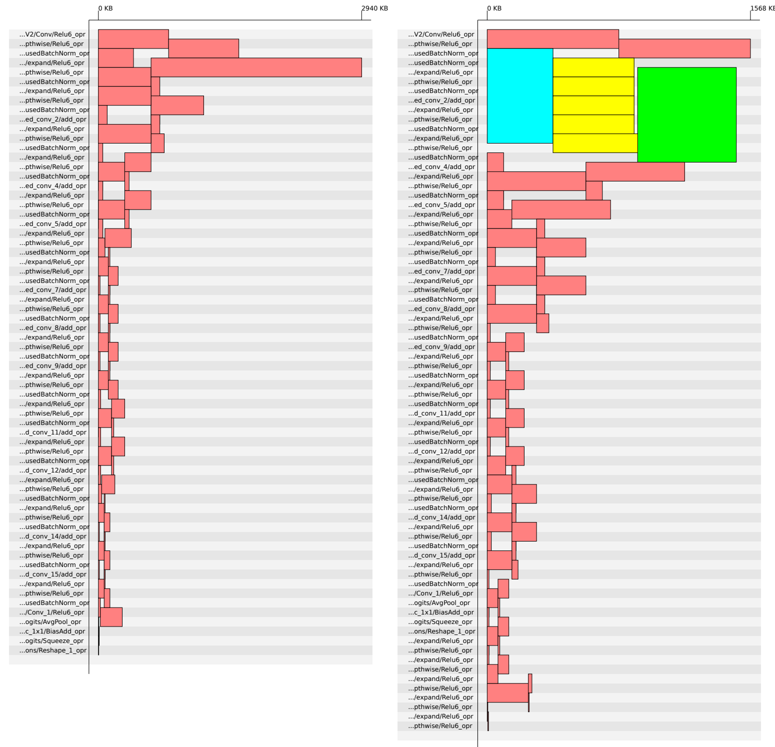


Figure 5.14: a, Original buffer allocation pattern of MobileNet v2 0.35 224. b, Optimised buffer allocation pattern after operation-splitting has been applied to the graph.

following depth-wise convolution into five parallel paths. This reduces the memory requirement of this part of the network so it is equal to the memory requirement of the following convolution operation. However this following convolution operation has equally sized input and output tensors so it cannot be optimised using operation splitting. The buffer allocation pattern for MobileNet v1 1.0 224 before and after operation-splitting is shown in Figure 5.13.

Table 5.4: Reduction in memory requirements of sequential published models

Model	**Original**	**DMO**		**Op Splitting**		
	PMR (KB)	PMR (KB)	Saving	PMR (KB)	Saving	Recomp
MobileNet v1 1.0 224	4704	3136	33.3%	3136	33.3%	5×10^6 (0.57%)
MobileNet v1 1.0 224 (8 bit)	1176	784	33.3%	784	33.3%	5×10^6 (0.57%)
MobileNet v1 0.25 224	1176	786	33.2%	980	16.7%	1.4×10^6 (0.25%)
MobileNet v1 0.25 128 (8 bit)	96	64	33.1%	64	33.3%	4.2×10^5 (0.99%)
MobileNet v2 0.35 224	2940	2352	20%	1568	46.7%	3×10^6 (0.73%)
MobileNet v2 1.0 224	5880	4704	20%	3528	40%	6.9×10^6 (0.47%)

Operation-splitting is more effective on MobileNet v2 variants than it is on v1 variants. Here the

peak memory requirement of MobileNet v2 0.35 224 is reduced by 46.7% and MobileNet v2 1.0 224 is reduced by 40%. The second convolution operation which defined the peak memory and the following depth-wise have been split into five chains. Reducing the memory required by these operations so it is lower than the memory required by the proceeding depth-wise convolution. Again this proceeding depth-wise operation has equal input and output tensors so cannot be optimised using operation splitting. The original and optimised buffer allocation patterns can be seen in Figure 5.14. This allows us to make a second important observation, that the memory saving of operation-splitting will be greater when input and output tensor sizes have a large disparity, and that no saving is possible if the input and output tensors are the same size.

We have shown that DMO produces its greatest savings when an operation has equally sized input and output tensors, and that operation-splitting produces its greatest savings when operations have very different sized input and output tensors, meaning these two methods are complimentary to each other. There are models which neither of these methods can optimise, but in some cases a poor result from one method will result in the best result of the other.

5.5.2 Connected Published Models

The densely connected models analysed showed lower memory savings in general than sequential models. Many of the operations within these models produce tensors which are used by more than one subsequent operation, the requirement for these tensor values to be held in memory for longer reduces opportunities to use DMO which clobbers the values of the input buffers. The connected nature of the models does not fundamentally affect operation-splitting in the same way as DMO but the savings can be expected to be lower. The peak memory requirement of connected networks is generally defined by a larger number buffers than sequential models. This means that if the size of a single buffer is reduced (which is what operation-splitting effectively achieves) this will have a lower impact on the overall peak memory requirement of the model.

In the worst cases ResNet 50 and the largest model analysed, Nasnet Mobile, their memory requirements could not be optimised at all using either DMO or operation splitting. Of the connected models analysed the greatest memory saving was observed for Inception ResNet, 34.5% saved using DMO and 19.3% saved using operation-splitting. However this saving was realised in the early input stage of the model which is sequential. The same is true of

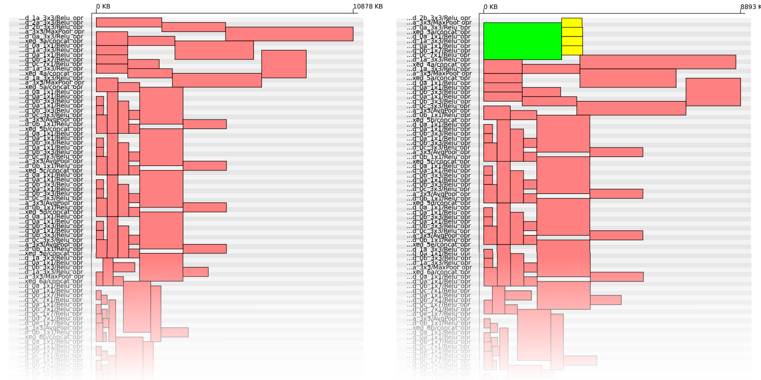


Figure 5.15: a, Inception v4 original buffer allocation pattern (only the first third of the model is shown for clarity). b, Buffer allocation pattern produced using operation-splitting, split tensors shown in yellow, final output of split block shown in green. Note that in this operation-splitting optimisation the input buffer is the start of the split, so is not shown in this figure.

Table 5.5: Reduction in memory requirements of connected published models

Model	**Original**	**DMO**		**Op Splitting**		
	PMR (KB)	PMR (KB)	Saving	PMR (KB)	Saving	Recomp
Inception v4 [136]	10879	10079	7.35%	8894	18.2%	2.7×10^7 (0.11%)
Inception ResNet v2 [136]	8399	5504	34.5%	6778	19.3%	2.8×10^5 (0.0007%)
Nasnet Mobile [165]	4540	4540	None	4541	None	-
DenseNet 121 [77]	8624	8232	4.55%	8624	None	-
ResNet 50 v2 [136]	10976	10976	None	10978	None	-

Inception v4, where it's greatest memory saving is achieved using operation splitting on the initial sequential stage of the model as shown in figure 5.15.

The 4.55% memory saving for DenseNet produced by DMO is an anomaly. In this instance the saving is not produced directly by diagonal optimisation but by a more optimal layout of non-overlapped buffers produced by the heap allocation strategy. In this case DMO has altered the order that these non-overlapped buffers are allocated so they take up less memory, see Figure 5.16. The modified heap allocation strategy used is a heuristic with no guarantee of optimality due to the NP-hard nature of the buffer allocation problem. It is possible that a more effective heap allocation strategy could produce a buffer pre-allocation pattern with this same peak memory requirement without the use of DMO.

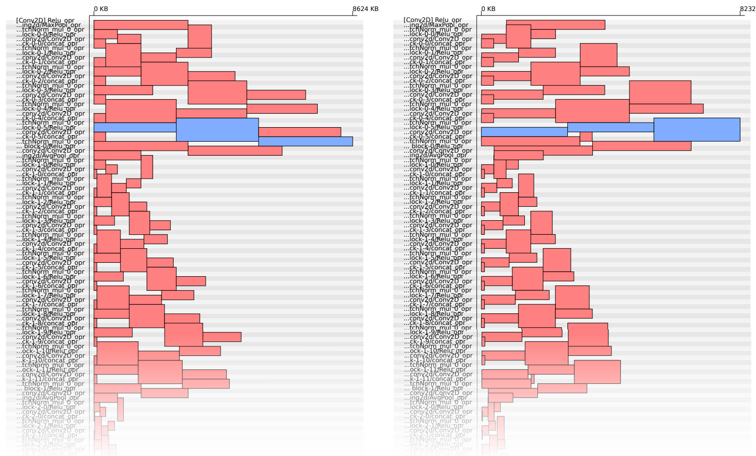


Figure 5.16: a, DenseNet original buffer allocation pattern (only the first fifth of the model is shown for clarity). b, Buffer allocation pattern optimised using DMO, Peak memory defining buffers are shown in blue. It can be seen that none of the peak memory defining buffers of the optimised pattern use DMO to overlap.

5.5.3 Cost-Mapping Models

The memory requirements of our proposed cost-mapping models were analysed when TFMin was first developed, and were discussed in Section 4.5.3. Both memory optimisation algorithms proposed in this chapter have been applied to these twelve models. The optimised memory pre-allocation patterns are approximately 50% smaller in the majority of cases, with a few models which are optimised significantly more or less.

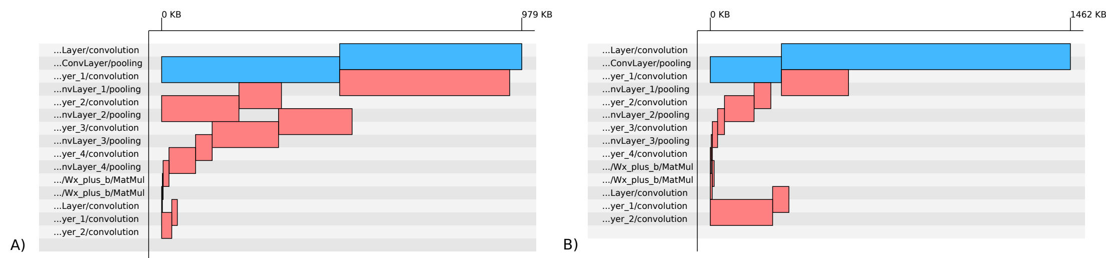


Figure 5.17: Original buffer pre-allocation patterns of: a, Model EncDec-F. b, Model EncDec-X. Intermediate buffers defining the peak memory requirement are highlighted in blue.

The Encoder-Decoder models all require more memory for inference than the smaller CNN models, which makes the optimisation of their memory use all the more important. We can see

from Table 5.6 that Encoder-Decoder models A-F are all optimised best using DMO resulting in almost a 50% saving. This is close to the maximum theoretical saving of this technique and is due to two equal sized tensors being produced by the first convolution and max-pooling layers, Figure 5.17.a.

Encoder-Decoder model X however does not have these equal sized large tensors so DMO is less effective saving only 19.8%. The peak memory requirement of this model is defined by output of the first convolution layer and the smaller output of the following max-pooling, Figure 5.17.b. This is a case which is particularly well suited to the operation-splitting technique, and this algorithm produces the greatest relative reduction in memory use of 61.6%.

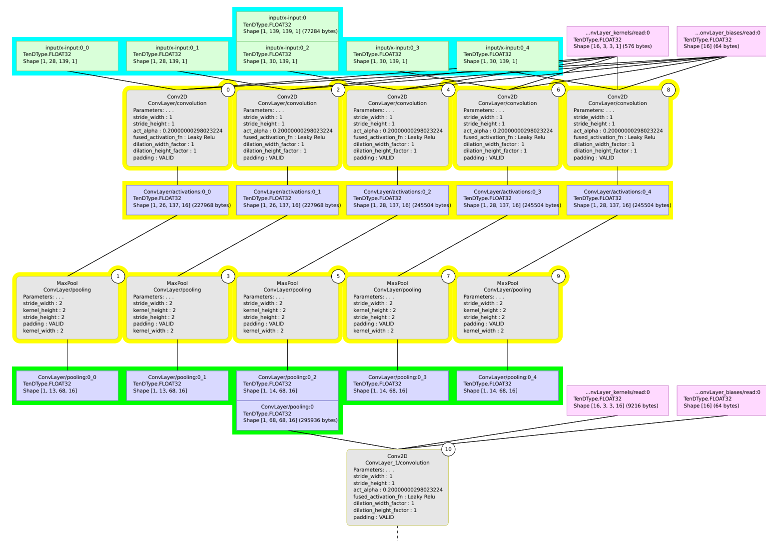


Figure 5.18: Initial section of the tensor graph of Encoder Decoder model X after modification by the operation-splitting memory optimiser. Showing the first two operations split into five parallel chains.

There is another benefit of using the operation-splitting optimiser on Encoder-Decoder model X. It is usual for models which are optimised using operation splitting for a small number of tensor elements to need recomputing, slightly slowing the inference process down. However this model does not require any of these re-computations even though it has been significantly optimised. The reason for this can be seen in the modified tensor graph produced by this optimiser shown in Figure 5.18. The first convolution and max-pooling operations have been split into 5 parallel chains, the elements of the second operation (max-pooling) have receptive fields which do not overlap. It has a kernel size of 2×2 and a stride of 2×2. This means that the outputs of the first

split-operation (convolution) do not need to overlap, therefore no re-computations are necessary.

Overall the two memory optimisation models have significantly reduced the RAM required for inference for all the Encoder-Decoder models. The complimentary nature of these algorithms whereby models which are not optimised well by one are optimised well by the other, has been clearly demonstrated.

Table 5.6: Reduction in required memory for inference of proposed cost-mapping models

Model	**Original**	**DMO**		**Op Splitting**		
	PMR (KB)	PMR (KB)	Saving	PMR (KB)	Saving	Recomp
Cnn-A	294	151	48.5%	276	6.1%	784 (0.48%)
Cnn-B	294	151	48.5%	276	6.1%	784 (0.46%)
Cnn-C	204	139	31.8%	147	28.0%	0 (0%)
Cnn-D	74	38	47.8%	69	6.1%	196 (0.41%)
Cnn-E	93	75	19.3%	41	56.3%	0 (0%)
EncDec-A	428	219	48.8%	406	5.0%	944 (0.39%)
EncDec-B	587	299	49.0%	561	4.3%	1104 (0.32%)
EncDec-C	979	497	49.2%	946	3.4%	1424 (0.24%)
EncDec-D	428	219	48.8%	406	5.0%	944 (0.39%)
EncDec-E	587	299	49.0%	561	4.3%	1104 (0.32%)
EncDec-F	979	497	49.2%	946	3.4%	1424 (0.24%)
EncDec-X	1462	1173	19.8%	561	61.6%	0 (0%)

5.6 Summary

Two novel methods to reduce the amount of RAM needed to perform inference using ML models have been presented. This reduction in the amount of volatile memory required is especially important in applications with limited computing resources, such as edge ML and on-board spacecraft.

The method of diagonal memory optimisation has been described and several methods presented to compute the critical buffer overlap O_s metric required to use this technique. Including a formal analytic solution which can quickly compute a lower bound for O_s. The technique of operation splitting has been described in detail and an algorithm presented which automatically applies it to any ML model. Both of these algorithms are available in the open-source TFMin software tool produced during this work [23].

Memory savings of up to 61.6% have been demonstrated on a range of ML models, the RAM required to perform inference using the smallest possible implementation of MobileNet has been reduced even further from 96 KB down to 64 KB. Both diagonal memory optimisation and operation-splitting will enable more efficient implementations of ML models in a range of application. In Edge ML applications larger models can be executed on micro-controller targets than currently possible, enabling smarter devices and or savings in power and cost. In space applications the RAM resources required by on-board ML processes has been reduced, easing the adoption of these solutions.

It is important to note that the required memory figures shown in Tables 5.4, 5.5, and 5.6 only include intermediate tensor values and not the weights of the model itself. In all models analysed the model weights require significantly more storage than the intermediate values themselves, MobileNet v1 0.25 224 for example has an optimised memory requirement of 786 KB but approximately 2.5 MB of weights. This would make it seem as though DMO is not much use in the real world, except that micro-controllers almost universally have much more non-volatile flash memory than volatile Static Random Access Memory (SRAM) [135]. The STM32F103xF from ST Microelectronics [132] is a commonly used ARM Cortex M3 micro-controller with 768 KB or 1 MB of program storage and 96 KB of SRAM. Using diagonal memory optimisation it becomes possible to execute the smallest MobileNet (v1 0.25 128 8bit) on this chip, however the weights of this model take 623 KB, 60.8% of the micro-controllers program memory. Similarly the Atmel AT32UC3C [6] used by the on-board computer of the ESAs ESEO mission [16] has at least four times more flash memory than SRAM across all its variants.

In early 2018 we started work building an automatic ML deployment tool because there were no open-source (or closed-source) tools we were aware of to perform this task. Concluding this work in 2020 there are now two other established tools, TFLμ [116] from Google and μTensor [5] from ARM. TFLμ places intermediate tensor buffers into a dedicated contiguous memory space in two possible ways, either pre-allocated or allocated on the fly. The core tool does not yet provide any pre-allocation algorithms since this is still an experimental feature. μTensor uses the platform 'malloc' to allocate intermediate tensors, so does not support buffer pre-allocation at this time.

Compared to these two tools TFMin has the most advanced memory optimisation capabilities,

and is able to deploy models onto smaller micro-controller targets. This is due in part to the memory optimisation techniques described in this chapter but also because it requires no run-time libraries to operate.

Opportunities and Progress

Table 5.7: Opportunities at the end of the cost mapping investigation.

Challenge	Solution
Is it possible to generate planetary rover cost-maps using ML?	• Encoder decoder models trained using supervised learning have been shown to be effective.
Are ML cost-mapping models feasible to use on radiation hardened LEON3 computers?	• Execution time has been quantified on representative hardware, and found to meet the 20 second requirement from Airbus. • Two novel memory optimisation techniques have been presented reducing the RAM requirement of cost-mapping models by up-to 61%.
Does a deployment process exist to implement this model within the flight software development process?	• The TFMin tool has been developed and released which generates prototype ANSI C code suitable for the flight software development process.

New Research Opportunities Identified in this Chapter

- Study the use of both DMO and Operation-Splitting concurrently to further reduce RAM needed by inference.

Chapter 6

Conclusion and Future Work

The rate at which orbital and surface missions are being launched to Mars has been gradually increasing in recent times. The most recent launch window in 2020 saw three nations dispatch robotic explorers, including the UAE's first Mars probe and two missions including rovers from China and the United States. Each of these surface missions present opportunities to trial and demonstrate increased levels of autonomy and novel technologies. The goal of this research was to demonstrate a novel cost-mapping algorithm, reducing the execution time of GNC processes on-board planetary rovers. Not only was this goal achieved, but an ML model generation process demonstrated which can trade off mapping accuracy for speed. This allows greater operational control of the GNC process, where the mapping process can be sped up when it is less critical. All This increase in speed translates into faster traverse times and lower demands on DSN capacity, ultimately resulting in greater mission capability. The specific rover mission this work targets is the proposed late 2020s Mars SFR currently under development by a consortium led by Airbus.

Work presented in this thesis has identified an opportunity to improve the performance of the autonomous GNC systems used on-board Mars rovers using the novel adoption of ML techniques. Novel ML cost-mapping models have been shown to produce equivalent quality maps while out performing existing state of the art algorithms used by the MER rovers. These performance gains have been verified on the LEON3 radiation hardened processor. Two novel memory optimisation algorithms have been presented which are able to significantly reduce the amount of RAM required to perform inference on CPU based targets. This approach has

the potential to improve the computational efficiency of autonomous rover GNC systems and thereby their traverse speeds and mission values. These algorithms not only impact the use of ML in space but are also relevant to the emerging field of 'Edge ML' on earth using small embedded micro-controllers.

This work will encourage and facilitate the use of on-board ML solutions on a wide variety of space missions, not only planetary rovers. This wider impact is expected to result from adoption of the TFMin deployment tool, enabling ML models to be analysed and deployed on flight processors. Allowing development to be conducted using a framework which has a realistic route for models to be deployed within flight software.

Chapter 3 described our investigation into ML cost-mapping models, and their use within the GNC system of planetary rovers. A range of CNN and Encoder-Decoder models (Table 3.1) were shown to be capable of performing this task, and their relative computational efficiency measured (Figure 3.21). It was found that computational efficiency was correlated to the number of cost-map cells estimated in a single pass. This finding influences the following work on model implementation, since it implies the largest model possible should be implemented to maximise efficiency. As well as meeting the original goal of this chapter, areas for future study were identified. Sparse cost-mapping models have the potential to improve the performance and suitability of these ML solutions for planetary rovers.

Chapter 4 presented a new software tool developed to facilitate the research presented in this thesis. The first published implementation of two state of the art memory optimisation algorithms is included to accelerate their adoption. This tool has been released to the community and is currently used by groups at Airbus studying on-board ML. TFMin started life as a tool developed to produce accurate timing results for ML models on the LEON family of radiation hardened space processors. During the course of this research and several design iterations this tool matured to include a wide range of functionality. Initial versions produced C++ 11 code with external dependencies while later versions moved to ANSI C with no dependencies. It is currently the only automated ML deployment tool which can target any ANSI C supported micro-controller or DSP. An open extensible framework is provided so users to customise the implementations generated for each layers, allowing optimised results for particular targets (Section 4.3.6. Airbus has expressed a desire to develop a commercial extension to this tool

including optimised LEON3/LEON4 layers which have been verified to ECSS standards.

Chapter 5 describes two novel algorithms performing a task which has to date had virtually no attention, the optimisation of RAM use by ML inference models. Until recently the amount of RAM required for inference has not been a limiting factor, however recent work implementing ML models on low power radiation hardened processors and terrestrial micro-controllers have hit these limits. The complimentary DMO (Section 5.4.2 and Operation-Splitting (Section 5.4.1) algorithms have been shown to reduce the RAM requirement of inference by up-to 46.7% in the case of MobileNet v2 (Table 5.4). These two algorithms represent the first aggressively optimised solutions to this problem, previous work has simply used general memory allocation techniques, not well suited to this task. These algorithms have been applied to the proposed cost-mapping ML models presented in Chapter 3, resulting in the memory requirement of the most efficient model (EncDec-X) being reduced by 61% (Table 5.6).

Table 6.1: Opportunities and proposed solutions presented in this thesis.

Challenge	**Solution**
Is it possible to generate planetary rover cost-maps using ML?	• Encoder decoder models trained using supervised learning have been shown to be effective.
Are ML cost-mapping models feasible to use on radiation hardened LEON3 computers?	• Execution time has been quantified on representative hardware, and found to meet the 20 second requirement from Airbus. • Two novel memory optimisation techniques have been presented reducing the RAM requirement of cost-mapping models by up-to 61%.
Does a deployment process exist to implement this model within the flight software development process?	• The TFMin tool has been developed and released which generates prototype ANSI C code suitable for the flight software development process.

6.1 Future Work

Chapter 3 presented a viable ML solution to the cost-mapping problem, however there are opportunities to increase its efficiency and practicality for an actual rover platform. The presented cost-map estimation models, all require dense input maps without unknown areas. This is a significant limitation given that maps generation on-board rovers are sparse and irregular due to limited depth perception and occlusions of their sensors. A valuable future investigation would develop ML cost-mapping models which process sparse input maps and generate cost-maps along with respective confidence levels. Such models could then be directly used within existing rover GNC stacks and trialled in simulations and on field trials.

The use of 2D scalar elevation maps without additional layers of information is a limitation the work presented. With access to more advanced baseline algorithms and terrain datasets, models could be developed incorporating a wider range of sensor data. The visual texture of terrain could be used alongside its geometrical shape for example. The biggest limitation of the ML cost-mapping technique described in this work is the need for an existing algorithm for supervised learning. This not only limits the ML solution to imitating this algorithm but also requires this algorithm to exist. A more ambitious study could use Reinforcement Learning and a high fidelity rover simulation to train a cost-mapping model from experience. This technique would remove the need for a baseline algorithm and potentially produce cost-maps which more accurately matched reality than traditional approaches.

The TFMin tool presented in Chapter 4 provides a useful framework for developing customised ML deployment systems. The open source version includes reference Op-Kernels which implement the most common ML operations, however more efficient algorithms do exist. Alongside the commercial development being considered by Airbus, the open source CMSIS NN library released by ARM includes a set of Neural Network (NN) layer implementations optimised for M series embedded processors. Adding this library to TFMin would result in far more optimal code for these targets. This would benefit terrestrial ‘Edge-ML’ application as well as lower cost space missions which use radiation tolerant ARM M series processors from Vorago Technologies [138], Microchip [108] and others.

A second limitation of this tool is its dependence upon the Tensorflow library, the only importers currently available are from TF sessions and exported TFL flatbuffers. Much cutting edge ML

research today is being done using simpler higher level frameworks such as PyTorch, which cannot be imported into TFMin at this time. The Open Neural Network Exchange (ONNX) standard was developed to provide a shared standard for communicating ML models between systems and has achieved widespread support among the frameworks used for research today. Adding an ONNX front end to TFMin would vastly increase its potential user base and impact within the ML community.

Chapter 5 presented the DMO and Operation-splitting memory optimisation algorithms which reduce the memory requirement of inference models. As discussed in Section 5.4.3 this algorithm is not compatible with multi-threaded execution in its current form. Given that all but the smallest micro-controllers contain more than a single core these days, addressing this limitation would allow this algorithm to be applied in a wide range of deployments.

Appendix A

Detailed Cost-Mapping Model Descriptions

A.1 Cnn-A Model

Layer	Output Size	Data Type	Details
Input	[1, 2601]	float 32	
Reshape	[1, 51, 51, 1]	float 32	
Conv2D	[1, 49, 49, 7]	float 32	Filter shape [3, 3, 1] Padding: VALID Stride: [1 1] Activation fn: Leaky Relu Bias added.
MaxPool	[1, 48, 48, 7]	float 32	Stride: [1 1] Padding: VALID Kernel size: [2 2]
Conv2D	[1, 46, 46, 7]	float 32	Filter shape [3, 3, 7] Padding: VALID Stride: [1 1] Activation fn: Leaky Relu Bias added.
MaxPool	[1, 23, 23, 7]	float 32	Stride: [2 2] Padding: VALID Kernel size: [2 2]

Conv2D	[1, 21, 21, 13]	float 32	Filter shape [3, 3, 7] Padding: VALID Stride: [1 1] Activation fn: Leaky Relu Bias added.
MaxPool	[1, 20, 20, 13]	float 32	Stride: [1 1] Padding: VALID Kernel size: [2 2]
Conv2D	[1, 18, 18, 13]	float 32	Filter shape [3, 3, 13] Padding: VALID Stride: [1 1] Activation fn: Leaky Relu Bias added.
MaxPool	[1, 9, 9, 13]	float 32	Stride: [2 2] Padding: VALID Kernel size: [2 2]
Conv2D	[1, 7, 7, 26]	float 32	Filter shape [3, 3, 13] Padding: VALID Stride: [1 1] Activation fn: Leaky Relu Bias added.
MaxPool	[1, 3, 3, 26]	float 32	Stride: [2 2] Padding: VALID Kernel size: [2 2]
Reshape	[1, 234]	float 32	
MatMul	[1, 117]	float 32	Weights [234, 117] Activation fn: Leaky Relu Bias added.
MatMul	[1, 117]	float 32	Weights [117, 117] Activation fn: Leaky Relu Bias added.
MatMul	[1, 1]	float 32	Weights [117, 1] Bias added.
Squeeze	[1]	float 32	
Output	[1]	float 32	

Table A.1: Detailed description of Cnn-A model.

A.2 Cnn-B Model

Layer	**Output Size**	**Data Type**	**Details**
Input	[1, 2601]	float 32	
Reshape	[1, 51, 51, 1]	float 32	
Conv2D	[1, 49, 49, 7]	float 32	Filter shape [3, 3, 1] Padding: VALID Stride: [1 1] Activation fn: Leaky Relu Bias added.
MaxPool	[1, 48, 48, 7]	float 32	Stride: [1 1] Padding: VALID Kernel size: [2 2]
Conv2D	[1, 46, 46, 7]	float 32	Filter shape [3, 3, 7] Padding: VALID Stride: [1 1] Activation fn: Leaky Relu Bias added.
MaxPool	[1, 23, 23, 7]	float 32	Stride: [2 2] Padding: VALID Kernel size: [2 2]
Conv2D	[1, 21, 21, 13]	float 32	Filter shape [3, 3, 7] Padding: VALID Stride: [1 1] Activation fn: Leaky Relu Bias added.
MaxPool	[1, 20, 20, 13]	float 32	Stride: [1 1] Padding: VALID Kernel size: [2 2]
Conv2D	[1, 18, 18, 26]	float 32	Filter shape [3, 3, 13] Padding: VALID Stride: [1 1] Activation fn: Leaky Relu Bias added.
MaxPool	[1, 9, 9, 26]	float 32	Stride: [2 2] Padding: VALID Kernel size: [2 2]
Reshape	[1, 2106]	float 32	

MatMul	[1, 117]	float 32	Weights [2106, 117] Activation fn: Leaky Relu Bias added.
MatMul	[1, 117]	float 32	Weights [117, 117] Activation fn: Leaky Relu Bias added.
MatMul	[1, 1]	float 32	Weights [117, 1] Bias added.
Squeeze	[1]	float 32	
Output	[1]	float 32	

Table A.2: Detailed description of Cnn-B model.

A.3 Cnn-C Model

Layer	**Output Size**	**Data Type**	**Details**
Input	[1, 2601]	float 32	
Reshape	[1, 51, 51, 1]	float 32	
Conv2D	[1, 49, 49, 3]	float 32	Filter shape [3, 3, 1] Padding: VALID Stride: [1 1] Activation fn: Leaky Relu Bias added.
MaxPool	[1, 48, 48, 3]	float 32	Stride: [1 1] Padding: VALID Kernel size: [2 2]
Conv2D	[1, 46, 46, 7]	float 32	Filter shape [3, 3, 3] Padding: VALID Stride: [1 1] Activation fn: Leaky Relu Bias added.
MaxPool	[1, 23, 23, 7]	float 32	Stride: [2 2] Padding: VALID Kernel size: [2 2]
Conv2D	[1, 21, 21, 10]	float 32	Filter shape [3, 3, 7] Padding: VALID Stride: [1 1] Activation fn: Leaky Relu Bias added.
MaxPool	[1, 20, 20, 10]	float 32	Stride: [1 1] Padding: VALID Kernel size: [2 2]
Conv2D	[1, 18, 18, 13]	float 32	Filter shape [3, 3, 10] Padding: VALID Stride: [1 1] Activation fn: Leaky Relu Bias added.
MaxPool	[1, 9, 9, 13]	float 32	Stride: [2 2] Padding: VALID Kernel size: [2 2]
Reshape	[1, 1053]	float 32	

MatMul	[1, 59]	float 32	Weights [1053, 59] Activation fn: Leaky Relu Bias added.
MatMul	[1, 59]	float 32	Weights [59, 59] Activation fn: Leaky Relu Bias added.
MatMul	[1, 1]	float 32	Weights [59, 1] Bias added.
Squeeze	[1]	float 32	
Output	[1]	float 32	

Table A.3: Detailed description of Cnn-C model.

A.4 Cnn-D Model

Layer	Output Size	Data Type	Details
Input	[1, 2601]	float 32	
Reshape	[1, 51, 51, 1]	float 32	
Conv2D	[1, 49, 49, 2]	float 32	Filter shape [3, 3, 1] Padding: VALID Stride: [1 1] Activation fn: Leaky Relu Bias added.
MaxPool	[1, 48, 48, 2]	float 32	Stride: [1 1] Padding: VALID Kernel size: [2 2]
Conv2D	[1, 46, 46, 2]	float 32	Filter shape [3, 3, 2] Padding: VALID Stride: [1 1] Activation fn: Leaky Relu Bias added.
MaxPool	[1, 23, 23, 2]	float 32	Stride: [2 2] Padding: VALID Kernel size: [2 2]
Conv2D	[1, 21, 21, 3]	float 32	Filter shape [3, 3, 2] Padding: VALID Stride: [1 1] Activation fn: Leaky Relu Bias added.
MaxPool	[1, 20, 20, 3]	float 32	Stride: [1 1] Padding: VALID Kernel size: [2 2]
Conv2D	[1, 18, 18, 10]	float 32	Filter shape [3, 3, 3] Padding: VALID Stride: [1 1] Activation fn: Leaky Relu Bias added.
MaxPool	[1, 9, 9, 10]	float 32	Stride: [2 2] Padding: VALID Kernel size: [2 2]
Reshape	[1, 810]	float 32	

MatMul	[1, 300]	float 32	Weights [810, 300] Activation fn: Leaky Relu Bias added.
MatMul	[1, 300]	float 32	Weights [300, 300] Activation fn: Leaky Relu Bias added.
MatMul	[1, 1]	float 32	Weights [300, 1] Bias added.
Squeeze	[1]	float 32	
Output	[1]	float 32	

Table A.4: Detailed description of Cnn-D model.

A.5 Cnn-E Model

Layer	Output Size	Data Type	Details
Input	[1, 2601]	float 32	
Reshape	[1, 51, 51, 1]	float 32	
Conv2D	[1, 49, 49, 3]	float 32	Filter shape [3, 3, 1] Padding: VALID Stride: [1 1] Activation fn: Leaky Relu Bias added.
MaxPool	[1, 24, 24, 3]	float 32	Stride: [2 2] Padding: VALID Kernel size: [2 2]
Conv2D	[1, 22, 22, 5]	float 32	Filter shape [3, 3, 3] Padding: VALID Stride: [1 1] Activation fn: Leaky Relu Bias added.
MaxPool	[1, 11, 11, 5]	float 32	Stride: [2 2] Padding: VALID Kernel size: [2 2]
Conv2D	[1, 9, 9, 13]	float 32	Filter shape [3, 3, 5] Padding: VALID Stride: [1 1] Activation fn: Leaky Relu Bias added.
MaxPool	[1, 4, 4, 13]	float 32	Stride: [2 2] Padding: VALID Kernel size: [2 2]
Reshape	[1, 208]	float 32	
MatMul	[1, 41]	float 32	Weights [208, 41] Activation fn: Leaky Relu Bias added.
MatMul	[1, 41]	float 32	Weights [41, 41] Activation fn: Leaky Relu Bias added.
MatMul	[1, 1]	float 32	Weights [41, 1] Bias added.

Squeeze	[1]	float 32	
Output	[1]	float 32	

Table A.5: Detailed description of Cnn-E model.

A.6 EncDec-A Model

Layer	Output Size	Data Type	Details
Input	[1, 3721]	float 32	
Reshape	[1, 61, 61, 1]	float 32	
Conv2D	[1, 59, 59, 7]	float 32	Filter shape [3, 3, 1] Padding: VALID Stride: [1 1] Activation fn: Leaky Relu Bias added.
MaxPool	[1, 58, 58, 7]	float 32	Stride: [1 1] Padding: VALID Kernel size: [2 2]
Conv2D	[1, 56, 56, 7]	float 32	Filter shape [3, 3, 7] Padding: VALID Stride: [1 1] Activation fn: Leaky Relu Bias added.
MaxPool	[1, 28, 28, 7]	float 32	Stride: [2 2] Padding: VALID Kernel size: [2 2]
Conv2D	[1, 26, 26, 13]	float 32	Filter shape [3, 3, 7] Padding: VALID Stride: [1 1] Activation fn: Leaky Relu Bias added.
MaxPool	[1, 25, 25, 13]	float 32	Stride: [1 1] Padding: VALID Kernel size: [2 2]
Conv2D	[1, 23, 23, 13]	float 32	Filter shape [3, 3, 13] Padding: VALID Stride: [1 1] Activation fn: Leaky Relu Bias added.
MaxPool	[1, 11, 11, 13]	float 32	Stride: [2 2] Padding: VALID Kernel size: [2 2]

Conv2D	[1, 9, 9, 26]	float 32	Filter shape [3, 3, 13] Padding: VALID Stride: [1 1] Activation fn: Leaky Relu Bias added.
MaxPool	[1, 4, 4, 26]	float 32	Stride: [2 2] Padding: VALID Kernel size: [2 2]
Reshape	[1, 416]	float 32	
MatMul	[1, 461]	float 32	Weights [416, 461] Activation fn: Leaky Relu Bias added.
MatMul	[1, 468]	float 32	Weights [461, 468] Activation fn: Leaky Relu Bias added.
Reshape	[1, 6, 6, 13]	float 32	
TransposedConv2D	[1, 11, 11, 1]	float 32	Filter shape [6, 6, 13] Padding: SAME Stride: [2 2] Bias added.
Output	[1, 11, 11, 1]	float 32	

Table A.6: Detailed description of EncDec-A model.

A.7 EncDec-B Model

Layer	Output Size	Data Type	Details
Input	[1, 5041]	float 32	
Reshape	[1, 71, 71, 1]	float 32	
Conv2D	[1, 69, 69, 7]	float 32	Filter shape [3, 3, 1] Padding: VALID Stride: [1 1] Activation fn: Leaky Relu Bias added.
MaxPool	[1, 68, 68, 7]	float 32	Stride: [1 1] Padding: VALID Kernel size: [2 2]
Conv2D	[1, 66, 66, 7]	float 32	Filter shape [3, 3, 7] Padding: VALID Stride: [1 1] Activation fn: Leaky Relu Bias added.
MaxPool	[1, 33, 33, 7]	float 32	Stride: [2 2] Padding: VALID Kernel size: [2 2]
Conv2D	[1, 31, 31, 13]	float 32	Filter shape [3, 3, 7] Padding: VALID Stride: [1 1] Activation fn: Leaky Relu Bias added.
MaxPool	[1, 30, 30, 13]	float 32	Stride: [1 1] Padding: VALID Kernel size: [2 2]
Conv2D	[1, 28, 28, 13]	float 32	Filter shape [3, 3, 13] Padding: VALID Stride: [1 1] Activation fn: Leaky Relu Bias added.
MaxPool	[1, 14, 14, 13]	float 32	Stride: [2 2] Padding: VALID Kernel size: [2 2]

Conv2D	[1, 12, 12, 26]	float 32	Filter shape [3, 3, 13] Padding: VALID Stride: [1 1] Activation fn: Leaky Relu Bias added.
MaxPool	[1, 6, 6, 26]	float 32	Stride: [2 2] Padding: VALID Kernel size: [2 2]
Reshape	[1, 936]	float 32	
MatMul	[1, 403]	float 32	Weights [936, 403] Activation fn: Leaky Relu Bias added.
MatMul	[1, 396]	float 32	Weights [403, 396] Activation fn: Leaky Relu Bias added.
Reshape	[1, 6, 6, 11]	float 32	
TransposedConv2D	[1, 11, 11, 13]	float 32	Filter shape [6, 6, 11] Padding: SAME Stride: [2 2] Activation fn: Leaky Relu Bias added.
TransposedConv2D	[1, 21, 21, 1]	float 32	Filter shape [11, 11, 13] Padding: SAME Stride: [2 2] Bias added.
Output	[1, 21, 21, 1]	float 32	

Table A.7: Detailed description of EncDec-B model.

A.8 EncDec-C Model

Layer	Output Size	Data Type	Details
Input	[1, 8281]	float 32	
Reshape	[1, 91, 91, 1]	float 32	
Conv2D	[1, 89, 89, 7]	float 32	Filter shape [3, 3, 1] Padding: VALID Stride: [1 1] Activation fn: Leaky Relu Bias added.
MaxPool	[1, 88, 88, 7]	float 32	Stride: [1 1] Padding: VALID Kernel size: [2 2]
Conv2D	[1, 86, 86, 7]	float 32	Filter shape [3, 3, 7] Padding: VALID Stride: [1 1] Activation fn: Leaky Relu Bias added.
MaxPool	[1, 43, 43, 7]	float 32	Stride: [2 2] Padding: VALID Kernel size: [2 2]
Conv2D	[1, 41, 41, 13]	float 32	Filter shape [3, 3, 7] Padding: VALID Stride: [1 1] Activation fn: Leaky Relu Bias added.
MaxPool	[1, 40, 40, 13]	float 32	Stride: [1 1] Padding: VALID Kernel size: [2 2]
Conv2D	[1, 38, 38, 13]	float 32	Filter shape [3, 3, 13] Padding: VALID Stride: [1 1] Activation fn: Leaky Relu Bias added.
MaxPool	[1, 19, 19, 13]	float 32	Stride: [2 2] Padding: VALID Kernel size: [2 2]

Conv2D	[1, 17, 17, 26]	float 32	Filter shape [3, 3, 13] Padding: VALID Stride: [1 1] Activation fn: Leaky Relu Bias added.
MaxPool	[1, 8, 8, 26]	float 32	Stride: [2 2] Padding: VALID Kernel size: [2 2]
Reshape	[1, 1664]	float 32	
MatMul	[1, 346]	float 32	Weights [1664, 346] Activation fn: Leaky Relu Bias added.
MatMul	[1, 360]	float 32	Weights [346, 360] Activation fn: Leaky Relu Bias added.
Reshape	[1, 6, 6, 10]	float 32	
TransposedConv2D	[1, 11, 11, 13]	float 32	Filter shape [6, 6, 10] Padding: SAME Stride: [2 2] Activation fn: Leaky Relu Bias added.
TransposedConv2D	[1, 21, 21, 7]	float 32	Filter shape [11, 11, 13] Padding: SAME Stride: [2 2] Activation fn: Leaky Relu Bias added.
TransposedConv2D	[1, 41, 41, 1]	float 32	Filter shape [21, 21, 7] Padding: SAME Stride: [2 2] Bias added.
Output	[1, 41, 41, 1]	float 32	

Table A.8: Detailed description of EncDec-C model.

A.9 EncDec-D Model

Layer	Output Size	Data Type	Details
Input	[1, 3721]	float 32	
Reshape	[1, 61, 61, 1]	float 32	
Conv2D	[1, 59, 59, 7]	float 32	Filter shape [3, 3, 1] Padding: VALID Stride: [1 1] Activation fn: Leaky Relu Bias added.
MaxPool	[1, 58, 58, 7]	float 32	Stride: [1 1] Padding: VALID Kernel size: [2 2]
Conv2D	[1, 56, 56, 7]	float 32	Filter shape [3, 3, 7] Padding: VALID Stride: [1 1] Activation fn: Leaky Relu Bias added.
MaxPool	[1, 28, 28, 7]	float 32	Stride: [2 2] Padding: VALID Kernel size: [2 2]
Conv2D	[1, 26, 26, 13]	float 32	Filter shape [3, 3, 7] Padding: VALID Stride: [1 1] Activation fn: Leaky Relu Bias added.
MaxPool	[1, 25, 25, 13]	float 32	Stride: [1 1] Padding: VALID Kernel size: [2 2]
Conv2D	[1, 23, 23, 13]	float 32	Filter shape [3, 3, 13] Padding: VALID Stride: [1 1] Activation fn: Leaky Relu Bias added.
MaxPool	[1, 11, 11, 13]	float 32	Stride: [2 2] Padding: VALID Kernel size: [2 2]

Conv2D	[1, 9, 9, 26]	float 32	Filter shape [3, 3, 13] Padding: VALID Stride: [1 1] Activation fn: Leaky Relu Bias added.
MaxPool	[1, 4, 4, 26]	float 32	Stride: [2 2] Padding: VALID Kernel size: [2 2]
Reshape	[1, 416]	float 32	
MatMul	[1, 202]	float 32	Weights [416, 202] Activation fn: Leaky Relu Bias added.
MatMul	[1, 216]	float 32	Weights [202, 216] Activation fn: Leaky Relu Bias added.
Reshape	[1, 6, 6, 6]	float 32	
TransposedConv2D	[1, 11, 11, 1]	float 32	Filter shape [6, 6, 6] Padding: SAME Stride: [2 2] Bias added.
Output	[1, 11, 11, 1]	float 32	

Table A.9: Detailed description of EncDec-D model.

A.10 EncDec-E Model

Layer	Output Size	Data Type	Details
Input	[1, 5041]	float 32	
Reshape	[1, 71, 71, 1]	float 32	
Conv2D	[1, 69, 69, 7]	float 32	Filter shape [3, 3, 1] Padding: VALID Stride: [1 1] Activation fn: Leaky Relu Bias added.
MaxPool	[1, 68, 68, 7]	float 32	Stride: [1 1] Padding: VALID Kernel size: [2 2]
Conv2D	[1, 66, 66, 7]	float 32	Filter shape [3, 3, 7] Padding: VALID Stride: [1 1] Activation fn: Leaky Relu Bias added.
MaxPool	[1, 33, 33, 7]	float 32	Stride: [2 2] Padding: VALID Kernel size: [2 2]
Conv2D	[1, 31, 31, 13]	float 32	Filter shape [3, 3, 7] Padding: VALID Stride: [1 1] Activation fn: Leaky Relu Bias added.
MaxPool	[1, 30, 30, 13]	float 32	Stride: [1 1] Padding: VALID Kernel size: [2 2]
Conv2D	[1, 28, 28, 13]	float 32	Filter shape [3, 3, 13] Padding: VALID Stride: [1 1] Activation fn: Leaky Relu Bias added.
MaxPool	[1, 14, 14, 13]	float 32	Stride: [2 2] Padding: VALID Kernel size: [2 2]

Conv2D	[1, 12, 12, 26]	float 32	Filter shape [3, 3, 13] Padding: VALID Stride: [1 1] Activation fn: Leaky Relu Bias added.
MaxPool	[1, 6, 6, 26]	float 32	Stride: [2 2] Padding: VALID Kernel size: [2 2]
Reshape	[1, 936]	float 32	
MatMul	[1, 202]	float 32	Weights [936, 202] Activation fn: Leaky Relu Bias added.
MatMul	[1, 216]	float 32	Weights [202, 216] Activation fn: Leaky Relu Bias added.
Reshape	[1, 6, 6, 6]	float 32	
TransposedConv2D	[1, 11, 11, 13]	float 32	Filter shape [6, 6, 6] Padding: SAME Stride: [2 2] Activation fn: Leaky Relu Bias added.
TransposedConv2D	[1, 21, 21, 1]	float 32	Filter shape [11, 11, 13] Padding: SAME Stride: [2 2] Bias added.
Output	[1, 21, 21, 1]	float 32	

Table A.10: Detailed description of EncDec-E model.

A.11 EncDec-F Model

Layer	Output Size	Data Type	Details
Input	[1, 8281]	float 32	
Reshape	[1, 91, 91, 1]	float 32	
Conv2D	[1, 89, 89, 7]	float 32	Filter shape [3, 3, 1] Padding: VALID Stride: [1 1] Activation fn: Leaky Relu Bias added.
MaxPool	[1, 88, 88, 7]	float 32	Stride: [1 1] Padding: VALID Kernel size: [2 2]
Conv2D	[1, 86, 86, 7]	float 32	Filter shape [3, 3, 7] Padding: VALID Stride: [1 1] Activation fn: Leaky Relu Bias added.
MaxPool	[1, 43, 43, 7]	float 32	Stride: [2 2] Padding: VALID Kernel size: [2 2]
Conv2D	[1, 41, 41, 13]	float 32	Filter shape [3, 3, 7] Padding: VALID Stride: [1 1] Activation fn: Leaky Relu Bias added.
MaxPool	[1, 40, 40, 13]	float 32	Stride: [1 1] Padding: VALID Kernel size: [2 2]
Conv2D	[1, 38, 38, 13]	float 32	Filter shape [3, 3, 13] Padding: VALID Stride: [1 1] Activation fn: Leaky Relu Bias added.
MaxPool	[1, 19, 19, 13]	float 32	Stride: [2 2] Padding: VALID Kernel size: [2 2]

Conv2D	[1, 17, 17, 26]	float 32	Filter shape [3, 3, 13] Padding: VALID Stride: [1 1] Activation fn: Leaky Relu Bias added.
MaxPool	[1, 8, 8, 26]	float 32	Stride: [2 2] Padding: VALID Kernel size: [2 2]
Reshape	[1, 1664]	float 32	
MatMul	[1, 202]	float 32	Weights [1664, 202] Activation fn: Leaky Relu Bias added.
MatMul	[1, 216]	float 32	Weights [202, 216] Activation fn: Leaky Relu Bias added.
Reshape	[1, 6, 6, 6]	float 32	
TransposedConv2D	[1, 11, 11, 13]	float 32	Filter shape [6, 6, 6] Padding: SAME Stride: [2 2] Activation fn: Leaky Relu Bias added.
TransposedConv2D	[1, 21, 21, 7]	float 32	Filter shape [11, 11, 13] Padding: SAME Stride: [2 2] Activation fn: Leaky Relu Bias added.
TransposedConv2D	[1, 41, 41, 1]	float 32	Filter shape [21, 21, 7] Padding: SAME Stride: [2 2] Bias added.
Output	[1, 41, 41, 1]	float 32	

Table A.11: Detailed description of EncDec-F model.

A.12 EncDec-X Model

Layer	Output Size	Data Type	Details
Input	[1, 19321]	float 32	
Reshape	[1, 139, 139, 1]	float 32	
Conv2D	[1, 137, 137, 7]	float 32	Filter shape [3, 3, 1] Padding: VALID Stride: [1 1] Activation fn: Leaky Relu Bias added.
MaxPool	[1, 68, 68, 7]	float 32	Stride: [2 2] Padding: VALID Kernel size: [2 2]
Conv2D	[1, 66, 66, 7]	float 32	Filter shape [3, 3, 7] Padding: VALID Stride: [1 1] Activation fn: Leaky Relu Bias added.
MaxPool	[1, 33, 33, 7]	float 32	Stride: [2 2] Padding: VALID Kernel size: [2 2]
Conv2D	[1, 31, 31, 13]	float 32	Filter shape [3, 3, 7] Padding: VALID Stride: [1 1] Activation fn: Leaky Relu Bias added.
MaxPool	[1, 15, 15, 13]	float 32	Stride: [2 2] Padding: VALID Kernel size: [2 2]
Conv2D	[1, 13, 13, 13]	float 32	Filter shape [3, 3, 13] Padding: VALID Stride: [1 1] Activation fn: Leaky Relu Bias added.
MaxPool	[1, 6, 6, 13]	float 32	Stride: [2 2] Padding: VALID Kernel size: [2 2]

Conv2D	[1, 4, 4, 26]	float 32	Filter shape [3, 3, 13] Padding: VALID Stride: [1 1] Activation fn: Leaky Relu Bias added.
Reshape	[1, 416]	float 32	
MatMul	[1, 807]	float 32	Weights [416, 807] Activation fn: Leaky Relu Bias added.
MatMul	[1, 864]	float 32	Weights [807, 864] Activation fn: Leaky Relu Bias added.
Reshape	[1, 12, 12, 6]	float 32	
TransposedConv2D	[1, 23, 23, 13]	float 32	Filter shape [12, 12, 6] Padding: SAME Stride: [2 2] Activation fn: Leaky Relu Bias added.
TransposedConv2D	[1, 45, 45, 13]	float 32	Filter shape [23, 23, 13] Padding: SAME Stride: [2 2] Activation fn: Leaky Relu Bias added.
TransposedConv2D	[1, 89, 89, 1]	float 32	Filter shape [45, 45, 13] Padding: SAME Stride: [2 2] Bias added.
Output	[1, 89, 89, 1]	float 32	

Table A.12: Detailed description of EncDec-X model.

Appendix B

Memory Optimisation Appendix

B.1 Tensorflow Lite Reference Operations

The following code has been taken from the open-source Tensorflow tool, they are from commit hash a0c6417 of version 2.1.0. The core **DepthwiseConv** function is taken from [**lite/kernels/internal/reference/depthwiseconv_float.h**]. Supporting structure **DepthwiseParams** and helper functions **MatchingDim**, **ComputePadding** and **Offset** are taken from [**lite/kernels/internal/types.h**] and [**lite/kernels/padding.h**].

This code can be used to analyse the complete thread of execution that is performed by this operation.

```
struct DepthwiseParams {
  PaddingType padding_type;
  PaddingValues padding_values;
  int16 stride_width;
  int16 stride_height;
  int16 dilation_width_factor;
  int16 dilation_height_factor;
  int16 depth_multiplier;
  // uint8 inference params.
  // TODO(b/65838351): Use smaller types if appropriate.
  int32 input_offset;
  int32 weights_offset;
  int32 output_offset;
  int32 output_multiplier;
  int output_shift;
  // uint8, etc, activation params.
  int32 quantized_activation_min;
```

```
    int32 quantized_activation_max;
    // float activation params.
    float float_activation_min;
    float float_activation_max;
  };

  // Get common shape dim, DCHECKing that they all agree.
  inline int MatchingDim(const RuntimeShape& shape1, int index1,
                         const RuntimeShape& shape2, int index2) {
    TFLITE_DCHECK_EQ(shape1.Dims(index1), shape2.Dims(index2));
    return shape1.Dims(index1);
  }

inline int ComputePadding(int stride, int dilation_rate, int in_size,
                          int filter_size, int out_size) {
  int effective_filter_size = (filter_size - 1) * dilation_rate + 1;
  int padding = ((out_size - 1) * stride +
                effective_filter_size - in_size) / 2;
  return padding > 0 ? padding : 0;
}

  inline int Offset(const RuntimeShape& shape, int i0, int i1, int i2, int i3) {
    TFLITE_DCHECK_EQ(shape.DimensionsCount(), 4);
    const int* dims_data = reinterpret_cast<const int*>(shape.DimsDataUpTo4D());
    TFLITE_DCHECK(i0 >= 0 && i0 < dims_data[0]);
    TFLITE_DCHECK(i1 >= 0 && i1 < dims_data[1]);
    TFLITE_DCHECK(i2 >= 0 && i2 < dims_data[2]);
    TFLITE_DCHECK(i3 >= 0 && i3 < dims_data[3]);
    return ((i0 * dims_data[1] + i1) * dims_data[2] + i2) * dims_data[3] + i3;
  }

  inline void DepthwiseConv(
  const DepthwiseParams& params, const RuntimeShape& input_shape,
  const float* input_data, const RuntimeShape& filter_shape,
  const float* filter_data, const RuntimeShape& bias_shape,
  const float* bias_data, const RuntimeShape& output_shape,
  float* output_data) {
    const int stride_width = params.stride_width;
    const int stride_height = params.stride_height;
    const int dilation_width_factor = params.dilation_width_factor;
    const int dilation_height_factor = params.dilation_height_factor;
    const int pad_width = params.padding_values.width;
    const int pad_height = params.padding_values.height;
    const int depth_multiplier = params.depth_multiplier;
    const float output_activation_min = params.float_activation_min;
    const float output_activation_max = params.float_activation_max;
    TFLITE_DCHECK_EQ(input_shape.DimensionsCount(), 4);
    TFLITE_DCHECK_EQ(filter_shape.DimensionsCount(), 4);
    TFLITE_DCHECK_EQ(output_shape.DimensionsCount(), 4);

    const int batches = MatchingDim(input_shape, 0, output_shape, 0);
    const int output_depth = MatchingDim(filter_shape, 3, output_shape, 3);
    const int input_height = input_shape.Dims(1);
```

```
    const int input_width = input_shape.Dims(2);
    const int input_depth = input_shape.Dims(3);
    const int filter_height = filter_shape.Dims(1);
    const int filter_width = filter_shape.Dims(2);
    const int output_height = output_shape.Dims(1);
    const int output_width = output_shape.Dims(2);
    TFLITE_DCHECK_EQ(output_depth, input_depth * depth_multiplier);
    TFLITE_DCHECK_EQ(bias_shape.FlatSize(), output_depth);

    for (int b = 0; b < batches; ++b) {
      for (int out_y = 0; out_y < output_height; ++out_y) {
        for (int out_x = 0; out_x < output_width; ++out_x) {
          for (int ic = 0; ic < input_depth; ++ic) {
            for (int m = 0; m < depth_multiplier; m++) {
              const int oc = m + ic * depth_multiplier;
              const int in_x_origin = (out_x * stride_width) - pad_width;
              const int in_y_origin = (out_y * stride_height) - pad_height;
              float total = 0.f;
              for (int filter_y = 0; filter_y < filter_height; ++filter_y) {
                for (int filter_x = 0; filter_x < filter_width; ++filter_x) {
                  const int in_x = in_x_origin + dilation_width_factor * filter_x;
                  const int in_y =
                      in_y_origin + dilation_height_factor * filter_y;
                  // If the location is outside the bounds of the input image,
                  // use zero as a default value.
                  if ((in_x >= 0) && (in_x < input_width) && (in_y >= 0) &&
                      (in_y < input_height)) {
                    float input_value =
                        input_data[Offset(input_shape, b, in_y, in_x, ic)];
                    float filter_value = filter_data[Offset(
                        filter_shape, 0, filter_y, filter_x, oc)];
                    total += (input_value * filter_value);
                  }
                }
              }
              float bias_value = 0.0f;
              if (bias_data) {
                bias_value = bias_data[oc];
              }
              output_data[Offset(output_shape, b, out_y, out_x, oc)] =
                  ActivationFunctionWithMinMax(total + bias_value,
                                               output_activation_min,
                                               output_activation_max);
            }
          }
        }
      }
    }
  }
```

Bibliography

[1] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, et al. Tensorflow: Large-scale machine learning on heterogeneous distributed systems. *arXiv preprint arXiv:1603.04467*, 2016.

[2] Gaisler Aeroflex. Dual-core leon3-ft sparc v8 processor. `http://www.gaisler.com/doc/gr712rc-datasheet.pdf`.

[3] Jörg Albertz. Albrecht meydenbauer-pioneer of photogrammetric documentation of the cultural heritage. *International Archives of Photogrammetry Remote Sensing and Spatial Information Sciences*, 34(5/C7):19–25, 2002.

[4] Andrew CM Allen, Christopher Langley, Raja Mukherji, Allen B Taylor, Manickam Umasuthan, and Timothy D Barfoot. Rendezvous lidar sensor system for terminal rendezvous, capture, and berthing to the international space station. In *Sensors and Systems for Space Applications II*, volume 6958, page 69580S. International Society for Optics and Photonics, 2008.

[5] ARM. utensor - test release. `https://github.com/uTensor/uTensor`.

[6] Ateml. 32-bit avr microcontroller. `http://ww1.microchip.com/downloads/en/DeviceDoc/doc32117.pdf`.

[7] Atmel. Rad-hard 32 bit sparc v8 processor at697f. `http://www.atmel.com/Images/doc7703.pdf`.

[8] TensorFlow Authors. Tensorflow models. `https://github.com/tensorflow/models`.

[9] Jimmy Lei Ba, Jamie Ryan Kiros, and Geoffrey E Hinton. Layer normalization. *arXiv preprint arXiv:1607.06450*, 2016.

[10] Vijay Badrinarayanan, Alex Kendall, and Roberto Cipolla. Segnet: A deep convolutional encoder-decoder architecture for image segmentation. *IEEE transactions on pattern analysis and machine intelligence*, 39(12):2481–2495, 2017.

[11] Ting Bai, Deren Li, Kaimin Sun, Yepei Chen, and Wenzhuo Li. Cloud detection for high-resolution satellite imagery using machine learning and multi-feature fusion. *Remote Sensing*, 8(9):715, 2016.

[12] Max Bajracharya, Mark W Maimone, and Daniel Helmick. Autonomy for mars rovers: Past, present, and future. *Computer*, 41(12):44–50, 2008.

[13] AJ Bakambu, M Nimelman, R Mukherji, and JW Tripp. Compact fast scanning lidar for planetary rover navigation. In *Proceedings of the International Symposium on Artificial Intelligence, Robotics and Automation in Space (I-SAIRAS), Turin, Italy*, pages 4–6, 2012.

[14] David Ball, Ben Upcroft, Gordon Wyeth, Peter Corke, Andrew English, Patrick Ross, Tim Patten, Robert Fitch, Salah Sukkarieh, and Andrew Bate. Vision-based obstacle detection and navigation for an agricultural robot. *Journal of field robotics*, 33(8):1107–1130, 2016.

[15] Shane Barratt. Interpnet: Neural introspection for interpretable deep learning. *arXiv preprint arXiv:1710.09511*, 2017.

[16] P Bartram, CP Bridges, D Bowman, and G Shirville. Software defined radio baseband processing for esa eseo mission. In *2017 IEEE Aerospace Conference*, pages 1–9. IEEE, 2017.

[17] Eric R Benton and EV Benton. Space radiation dosimetry in low-earth orbit and beyond. *Nuclear Instruments and Methods in Physics Research Section B: Beam Interactions with Materials and Atoms*, 184(1-2):255–294, 2001.

[18] Richard W Berger, Devin Bayles, Ronald Brown, Scott Doyle, Abbas Kazemzadeh, Ken Knowles, D Moser, J Rodgers, B Saari, D Stanley, et al. The rad750/sup tm/-a radiation hardened powerpc/sup tm/processor for high performance spaceborne applications. In *2001 IEEE Aerospace Conference Proceedings (Cat. No. 01TH8542)*, volume 5, pages 2263–2272. IEEE, 2001.

[19] Jo Bermyn. Proba-project for on-board autonomy. *Air & Space Europe*, 2(1):70–76, 2000.

[20] Kevin Berry, Brian Sutter, Alex May, Ken Williams, Brent W Barbee, Mark Beckman, and Bobby Williams. Osiris-rex touch-and-go (tag) mission design and analysis. 2013.

[21] Jeffrey J Biesiadecki and Mark W Maimone. The mars exploration rover surface mobility flight software driving ambition. In *2006 IEEE Aerospace Conference*, pages 15–pp. IEEE, 2006.

[22] P Blacker, CP Bridges, and S Hadfield. Rapid prototyping of deep learning models on radiation hardened cpus. In *2019 NASA/ESA Conference on Adaptive Hardware and Systems (AHS)*, pages 25–32. IEEE, 2019.

[23] Pete Blacker. Github repository of the tfmin code generation tool. `https://github.com/PeteBlackerThe3rd/TFMin`.

[24] Pete Blacker. Visual memory tracer. `https://github.com/PeteBlackerThe3rd/VisualMemoryTracer`.

[25] Harold Borkan. Radiation hardening of cmos technologies-an overview. *IEEE Transactions on Nuclear Science*, 24(6):2043–2046, 1977.

[26] John L Callas, Matthew P Golombek, and Abigail A Fraeman. Mars exploration rover opportunity end of mission report. Technical report, Pasadena, CA: Jet Propulsion Laboratory, National Aeronautics and Space , 2019.

[27] Tanner Campbell. A deep learning approach to autonomous relative terrain navigation. 2017.

[28] Joseph Carsten, Arturo Rankin, Dave Ferguson, and Anthony Stentz. Global path planning on board the mars exploration rovers. In *2007 IEEE Aerospace Conference*, pages 1–11. IEEE, 2007.

[29] Rebecca Castano, Kiri L Wagstaff, Steve Chien, Timothy M Stough, and Benyang Tang. On-board analysis of uncalibrated data for a spacecraft at mars. In *Proceedings of the 13th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 922–930, 2007.

[30] Liang-Chieh Chen, Yukun Zhu, George Papandreou, Florian Schroff, and Hartwig Adam. Encoder-decoder with atrous separable convolution for semantic image segmentation. In *Proceedings of the European conference on computer vision (ECCV)*, pages 801–818, 2018.

[31] Xieyuanli Chen, Hui Zhang, Huimin Lu, Junhao Xiao, Qihang Qiu, and Yi Li. Robust slam system based on monocular vision and lidar for robotic urban search and rescue. In *2017 IEEE International Symposium on Safety, Security and Rescue Robotics (SSRR)*, pages 41–47. IEEE, 2017.

[32] Wei-Kai Cheng, Po-Yuan Shen, and Xin-Lun Li. Retention-aware dram auto-refresh scheme for energy and performance efficiency. *Micromachines*, 10(9):590, 2019.

[33] Steve Chien, Richard Doyle, Ashley Gerard Davies, Ari Jonsson, and Ralph Lorenz. The future of ai in space. *IEEE Intelligent Systems*, 21(4):64–69, 2006.

[34] Valérie Ciarletti, Stephen Clifford, Dirk Plettemeier, Alice Le Gall, Yann Hervé, Sophie Dorizon, Cathy Quantin-Nataf, Wolf-Stefan Benedix, Susanne Schwenzer, Elena Pettinelli, et al. The wisdom radar: unveiling the subsurface beneath the exomars rover and identifying the best locations for drilling. *Astrobiology*, 17(6-7):565–584, 2017.

[35] Dan Ciregan, Ueli Meier, and Jürgen Schmidhuber. Multi-column deep neural networks for image classification. In *2012 IEEE conference on computer vision and pattern recognition*, pages 3642–3649. IEEE, 2012.

[36] AJ Coates, Ralf Jaumann, AD Griffiths, CE Leff, Nicole Schmitz, J-L Josset, Gerhard Paar, Matthew Gunn, Ernst Hauber, Claire Rachel Cousins, et al. The pancam instrument for the exomars rover. *Astrobiology*, 17(6-7):511–541, 2017.

[37] Derek Coleman, Patrick Arnold, Stephanie Bodoff, Chris Dollin, Helena Gilchrist, Fiona Hayes, and Paul Jeremaes. *Object-oriented development: the fusion method*. Prentice-Hall, Inc., 1994.

[38] Torch Contributors. Torchscript. `https://pytorch.org/docs/stable/jit.html`.

[39] Luc Devroye and Paul Kruszewski. A note on the horton-strahler number for random trees. *Information processing letters*, 52(3):155–159, 1994.

[40] PE Dodd, MR Shaneyfelt, JR Schwank, and JA Felix. Current and future challenges in radiation effects on cmos electronics. *IEEE Transactions on Nuclear Science*, 57(4):1747–1763, 2010.

[41] Daxiang Dong, Hua Wu, Wei He, Dianhai Yu, and Haifeng Wang. Multi-task learning for multiple language translation. In *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, pages 1723–1732, 2015.

[42] Alexey Dosovitskiy, Philipp Fischer, Eddy Ilg, Philip Hausser, Caner Hazirbas, Vladimir Golkov, Patrick Van Der Smagt, Daniel Cremers, and Thomas Brox. Flownet: Learning optical flow with convolutional networks. In *Proceedings of the IEEE international conference on computer vision*, pages 2758–2766, 2015.

[43] Paul Drews, Grady Williams, Brian Goldfain, Evangelos A Theodorou, and James M Rehg. . *IEEE Robotics and Automation Letters*, 4(2):1564–1571, 2019.

[44] Alberto Elfes. Using occupancy grids for mobile robot perception and navigation. *Computer*, 22(6):46–57, 1989.

[45] Elisabete Fernandes, Pedro Costa, José Lima, and Germano Veiga. Towards an orientation enhanced astar algorithm for robotic navigation. In *2015 IEEE International Conference on Industrial Technology (ICIT)*, pages 3320–3325. IEEE, 2015.

[46] Paolo Ferri and E Sorensen. Automated mission operations for rosetta. In *Proceeding of the Fifth International Symposium on Space Mission Operations and Ground Data System: SpaceOps*, volume 98. Citeseer, 1998.

[47] David Fofi, Tadeusz Sliwa, and Yvon Voisin. A comparative survey on invisible structured light. In *Machine vision applications in industrial inspection XII*, volume 5303, pages 90–98. International Society for Optics and Photonics, 2004.

[48] The European Cooperation for Space Standardization. *ECSS-E-ST-40C Software Engineering*. ESA, 2009.

[49] The European Cooperation for Space Standardization. *ECSS-Q-ST-80C Software Product Assurance*. ESA, 2009.

[50] Huan Fu, Mingming Gong, Chaohui Wang, Kayhan Batmanghelich, and Dacheng Tao. Deep ordinal regression network for monocular depth estimation. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 2002–2011, 2018.

[51] Cobham Gaisler. Gr-cpci-gr740 quad-core leon4ft development board. `https://www.gaisler.com/index.php/products/boards/gr-cpci-gr740`.

[52] Cobham Gaisler. Gr740 quad-processor leon4ft system-on-chip overview. `http://gaisler.com/doc/gr740/GR740-OVERVIEW.pdf`.

[53] Jiri Gaisler and Konrad Eisele. Bcc-bare-c cross-compiler users manual. *Aeroflex Gaisler AB*, 2012.

[54] David Gallup, Jan-Michael Frahm, Philippos Mordohai, and Marc Pollefeys. Variable baseline/resolution stereo. In *2008 IEEE conference on computer vision and pattern recognition*, pages 1–8. IEEE, 2008.

[55] Alan D George and Christopher M Wilson. Onboard processing with hybrid and reconfigurable computing on small satellites. *Proceedings of the IEEE*, 106(3):458–470, 2018.

[56] Spyros Gidaris and Nikos Komodakis. Detect, replace, refine: Deep structured prediction for pixel wise labeling. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 5248–5257, 2017.

[57] Gaël Guennebaud, Benoit Jacob, et al. Eigen. *URl: http://eigen. tuxfamily. org*, 2010.

[58] Antonio Gulli and Sujit Pal. *Deep learning with Keras*. Packt Publishing Ltd, 2017.

[59] N Haddad, R Brown, T Cronauer, and H Phan. Radiation hardened cots-based 32-bit microprocessor. In *1999 Fifth European Conference on Radiation and Its Effects on Components and Systems. RADECS 99 (Cat. No. 99TH8471)*, pages 593–597. IEEE, 1999.

[60] Rostam Affendi Hamzah and Haidi Ibrahim. Literature survey on stereo vision disparity map algorithms. *Journal of Sensors*, 2016, 2016.

[61] Song Han, Xingyu Liu, Huizi Mao, Jing Pu, Ardavan Pedram, Mark A Horowitz, and William J Dally. Eie: efficient inference engine on compressed deep neural network. *ACM SIGARCH Computer Architecture News*, 44(3):243–254, 2016.

[62] Brian D Harrington and Chris Voorhees. The challenges of designing the rocker-bogie suspension for the mars exploration rover. 2004.

[63] Richard D Harris, Steven S McClure, Bernard G Rax, Robin W Evans, and Insoo Jun. Comparison of tid effects in space-like variable dose rates and constant dose rates. *IEEE Transactions on Nuclear Science*, 55(6):3088–3095, 2008.

[64] Peter E Hart, Nils J Nilsson, and Bertram Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE transactions on Systems Science and Cybernetics*, 4(2):100–107, 1968.

[65] Yihui He, Ji Lin, Zhijian Liu, Hanrui Wang, Li-Jia Li, and Song Han. Amc: Automl for model compression and acceleration on mobile devices. In *Proceedings of the European Conference on Computer Vision (ECCV)*, pages 784–800, 2018.

[66] James R Heirtzler. The future of the south atlantic anomaly and implications for radiation damage in space. *Journal of Atmospheric and Solar-Terrestrial Physics*, 64(16):1701–1708, 2002.

[67] Daniel Helmick, Anelia Angelova, and Larry Matthies. Terrain adaptive navigation for planetary rovers. *Journal of Field Robotics*, 26(4):391–410, 2009.

[68] Nicholas J Higham. The accuracy of floating point summation. *SIAM Journal on Scientific Computing*, 14(4):783–799, 1993.

[69] Kelsey Hightower, Brendan Burns, and Joe Beda. *Kubernetes: up and running: dive into the future of infrastructure*. " O'Reilly Media, Inc.", 2017.

[70] Geoffrey E Hinton and Ruslan R Salakhutdinov. Reducing the dimensionality of data with neural networks. *science*, 313(5786):504–507, 2006.

[71] Gerard J Holzmann. The power of 10: Rules for developing safety-critical code. *Computer*, 39(6):95–99, 2006.

[72] Andrew G Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *arXiv preprint arXiv:1704.04861*, 2017.

[73] Thomas M Howard, Arin Morfopoulos, Jack Morrison, Yoshiaki Kuwata, Carlos Villalpando, Larry Matthies, and Michael McHenry. Enabling continuous planetary rover navigation through fpga stereo and visual odometry. In *2012 IEEE Aerospace Conference*, pages 1–9. IEEE, 2012.

[74] Renyu Hu, A Anthony Bloom, Peter Gao, Charles E Miller, and Yuk L Yung. Hypotheses for near-surface exchange of methane on mars. *Astrobiology*, 16(7):539–550, 2016.

[75] Itay Hubara, Matthieu Courbariaux, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. Binarized neural networks. In *Proceedings of the 30th international conference on neural information processing systems*, pages 4114–4122. Citeseer, 2016.

[76] Karl Iagnemma, Shinwoo Kang, Hassan Shibly, and Steven Dubowsky. Online terrain parameter estimation for wheeled mobile robots with application to planetary rovers. *IEEE transactions on robotics*, 20(5):921–927, 2004.

[77] Forrest Iandola, Matt Moskewicz, Sergey Karayev, Ross Girshick, Trevor Darrell, and Kurt Keutzer. Densenet: Implementing efficient convnet descriptor pyramids. *arXiv preprint arXiv:1404.1869*, 2014.

[78] Forrest N Iandola, Song Han, Matthew W Moskewicz, Khalid Ashraf, William J Dally, and Kurt Keutzer. Squeezenet: Alexnet-level accuracy with 50x fewer parameters and¡ 0.5 mb model size. *arXiv preprint arXiv:1602.07360*, 2016.

[79] Max Jaderberg, Andrea Vedaldi, and Andrew Zisserman. Speeding up convolutional neural networks with low rank expansions. *arXiv preprint arXiv:1405.3866*, 2014.

[80] Manpreet Kaur Jaswal, Debjyoti Mallik, and Manjit Kaur. Radiation hardened seu tolerant reed solomon encoder and decoder. In *2016 3rd International Conference on Signal Processing and Integrated Networks (SPIN)*, pages 417–420. IEEE, 2016.

[81] IP Karachevtseva, AA Kokhanov, NA Kozlova, and Zh F Rodionova. Cartography of the soviet lunokhods routes on the moon. In *Planetary Cartography and GIS*, pages 263–278. Springer, 2019.

[82] P Kidwell. Journey to the moon: the history of the apollo guidance computer. *IEEE Annals of the History of Computing*, 21(1):78–79, 1999.

[83] Jong Hwan Ko, Duckhwan Kim, Taesik Na, Jaeha Kung, and Saibal Mukhopadhyay. Adaptive weight compression for memory-efficient neural networks. In *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2017*, pages 199–204. IEEE, 2017.

[84] Kurt Konolige, Motilal Agrawal, Robert C Bolles, Cregg Cowan, Martin Fischler, and Brian Gerkey. Outdoor mapping and navigation using stereo vision. In *Experimental Robotics*, pages 179–190. Springer, 2008.

[85] Oleg Korablev, Ann Carine Vandaele, Franck Montmessin, Anna A Fedorova, Alexander Trokhimovskiy, François Forget, Franck Lefèvre, Frank Daerden, Ian R Thomas, Loïc Trompet, et al. No detection of methane on mars from early exomars trace gas orbiter observations. *Nature*, 568(7753):517–520, 2019.

[86] Daniel G Kubitschek, Nickolaos Mastrodemos, Robert A Werner, Brian M Kennedy, Stephen P Synnott, George W Null, Shyam Bhaskaran, Joseph E Riedel, and Andrew T Vaughan. Deep impact autonomous navigation: the trials of targeting the unknown. 2006.

[87] Fadi J Kurdahi and Alice C Parker. Real: A program for register allocation. In *Proceedings of the 24th ACM/IEEE Design Automation Conference*, pages 210–215, 1987.

[88] Liangzhen Lai, Naveen Suda, and Vikas Chandra. Cmsis-nn: Efficient neural network kernels for arm cortex-m cpus. *arXiv preprint arXiv:1801.06601*, 2018.

[89] David J Lary, Gebreab K Zewdie, Xun Liu, Daji Wu, Estelle Levetin, Rebecca J Allee, Nabin Malakar, Annette Walker, Hamse Mussa, Antonio Mannino, et al. Machine learning applications for earth observation. *Earth observation open science and innovation*, 165, 2018.

[90] Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*, page 75. IEEE Computer Society, 2004.

[91] Alexander Lavin. Optimized mission planning for planetary exploration rovers. *arXiv preprint arXiv:1511.00195*, 2015.

[92] Chao Li, Yi Yang, Min Feng, Srimat Chakradhar, and Huiyang Zhou. Optimizing memory efficiency for deep convolutional neural networks on gpus. In *SC'16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 633–644. IEEE, 2016.

[93] Guilin Liu, Fitsum A Reda, Kevin J Shih, Ting-Chun Wang, Andrew Tao, and Bryan Catanzaro. Image inpainting for irregular holes using partial convolutions. In *Proceedings of the European Conference on Computer Vision (ECCV)*, pages 85–100, 2018.

[94] Shusen Liu, Bhavya Kailkhura, Donald Loveland, and Yong Han. Generative counterfactual introspection for explainable deep learning. *arXiv preprint arXiv:1907.03077*, 2019.

[95] Andrea Lodi, Silvano Martello, and Daniele Vigo. Approximation algorithms for the oriented two-dimensional bin packing problem. *European Journal of Operational Research*, 112(1):158–166, 1999.

[96] David A Lorenz, Ryan Olds, Alexander May, Courtney Mario, Mark E Perry, Eric E Palmer, and Michael Daly. Lessons learned from osiris-rex autonomous navigation using natural feature tracking. In *2017 IEEE Aerospace Conference*, pages 1–12. IEEE, 2017.

[97] David V Lu, Dave Hershberger, and William D Smart. Layered costmaps for context-sensitive navigation. In *2014 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 709–715. IEEE, 2014.

[98] Mark Maimone, Yang Cheng, and Larry Matthies. Two years of visual odometry on the mars exploration rovers. *Journal of Field Robotics*, 24(3):169–186, 2007.

[99] J Maki, D Thiessen, A Pourangi, P Kobzeff, T Litwin, L Scherr, S Elliott, A Dingizian, and M Maimone. The mars science laboratory engineering cameras. *Space science reviews*, 170(1-4):77–93, 2012.

[100] Michal C Malin, Michael A Ravine, Michael A Caplinger, F Tony Ghaemi, Jacob A Schaffner, Justin N Maki, James F Bell III, James F Cameron, William E Dietrich, Kenneth S Edgett, et al. The mars science laboratory (msl) mast cameras and descent imager: investigation and instrument descriptions. *Earth and Space Science*, 4(8):506–539, 2017.

[101] Jacob Manning, David Langerman, Barath Ramesh, Evan Gretok, Christopher Wilson, Alan George, James MacKinnon, and Gary Crum. Machine-learning space applications on smallsat platforms with tensorflow. In *Proceedings of the 32nd Annual AIAA/USU Conference on Small Satellites, Logan, UT, USA*, pages 4–9, 2018.

[102] J Matijevic. Sojourner the mars pathfinder microrover flight experiment. 1997.

[103] S Maurice, RC Wiens, M Saccoccio, B Barraclough, O Gasnault, O Forni, N Mangold, David Baratoux, S Bender, G Berger, et al. The chemcam instrument suite on the mars science laboratory (msl) rover: Science objectives and mast unit description. *Space science reviews*, 170(1-4):95–166, 2012.

[104] Nikolaus Mayer, Eddy Ilg, Philip Hausser, Philipp Fischer, Daniel Cremers, Alexey Dosovitskiy, and Thomas Brox. A large dataset to train convolutional networks for disparity, optical flow, and scene flow estimation. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 4040–4048, 2016.

[105] TH McConnochie, JF Bell III, D Savransky, G Mehall, M Caplinger, PR Christensen, L Cherednik, K Bender, and A Dombovari. Calibration and in-flight performance of the mars odyssey thermal emission imaging system visible imaging subsystem (themis vis). *Journal of Geophysical Research: Planets*, 111(E6), 2006.

[106] Kevin McManamon, Richard Lancaster, and Nuno Silva. Exomars rover vehicle perception system architecture and test results. In *Proceedings of the 12th Symposium on Advanced Space Technologies in Robotics and Automation, Noordwijk, The Netherlands*, pages 15–17, 2013.

[107] Stéphane Michaud, Andreas Gibbesch, Thomas Thüer, Ambroise Krebs, Christopher Lee, B Despont, Bernd Schäfer, and Richard Slade. Development of the exomars chassis and locomotion subsystem. In *Proc. of The 9th International Symposium on Artificial Intelligence, Robotics and Automation in Space (iSAIRAS)*. Eidgenössische Technische Hochschule Zürich, Autonomous Systems Lab, 2008.

[108] Microchip. Microchip space solutions. `file:///home/user/Downloads/Space_Solutions_Brochure.pdf`.

[109] Gareth Llewellyn Keith Morgan, Jian Guo Liu, and Hongshi Yan. Precise subpixel disparity measurement from very narrow baseline stereo. *IEEE Transactions on Geoscience and Remote Sensing*, 48(9):3424–3433, 2010.

[110] Shiva Nejati, Stefano Di Alesio, Mehrdad Sabetzadeh, and Lionel Briand. Modeling and analysis of cpu usage in safety-critical embedded systems to support stress testing. In *International Conference on Model Driven Engineering Languages and Systems*, pages 759–775. Springer, 2012.

[111] Nicholas Nethercote and Julian Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *ACM Sigplan notices*, volume 42, pages 89–100. ACM, 2007.

[112] Martha V O'Bryan, Kenneth A LaBel, Ray L Ladbury, Christian Poivey, JW Howard, Robert A Reed, Scott D Kniffin, Stephen P Buchner, John P Bings, Jeff L Titus, et al. Current single event effects and radiation damage results for candidate spacecraft electronics. In *IEEE Radiation Effects Data Workshop*, pages 82–105. IEEE, 2002.

[113] Angus Pacala, Tianyue Yu, and Louay Eldada. Cost-effective lidar sensor for multi-signal detection, weak signal detection and signal disambiguation and method of using same, July 31 2014. US Patent App. 14/165,566.

[114] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. In *Advances in neural information processing systems*, pages 8026–8037, 2019.

[115] Mukund R Patel. *Spacecraft power systems*. CRC press, 2004.

[116] Daniel S Pete W. *TinyML*. O'Reilly, 2019.

[117] Mihail Pivtoraiko, Thomas M Howard, I Nesnas, and Alonzo Kelly. Field experiments in rover navigation via model-based trajectory generation and nonholonomic motion planning in state lattices. In *Proceedings of the 9th International Symposium on Artificial Intelligence, Robotics, and Automation in Space*, pages 25–29, 2008.

[118] Vaughan Pratt. Anatomy of the pentium bug. In *Colloquium on Trees in Algebra and Programming*, pages 97–107. Springer, 1995.

[119] Arturo Rankin, Mark Maimone, Jeffrey Biesiadecki, Nikunj Patel, Dan Levine, and Olivier Toupet. Driving curiosity: Mars rover mobility trends during the first seven years. In *2020 IEEE Aerospace Conference*, pages 1–19. IEEE, 2020.

[120] Patrick Ross, Andrew English, David Ball, Ben Upcroft, and Peter Corke. Online novelty-based visual obstacle detection for field robotics. In *2015 IEEE International Conference on Robotics and Automation (ICRA)*, pages 3935–3940. IEEE, 2015.

[121] Stephane Ruel, Tim Luu, and Andrew Berube. Space shuttle testing of the tridar 3d rendezvous and docking sensor. *Journal of Field robotics*, 29(4):535–553, 2012.

[122] Tim Salimans and Durk P Kingma. Weight normalization: A simple reparameterization to accelerate training of deep neural networks. In *Advances in neural information processing systems*, pages 901–909, 2016.

[123] Arash Sangari and William Sethares. Convergence analysis of two loss functions in soft-max regression. *IEEE Transactions on Signal Processing*, 64(5):1280–1288, 2015.

[124] PS Schenker, LF Sword, J Ganino, B Bickler, GS Hickey, and DK Brown. Lightweight rovers for mars science exploration aqd sample return.

[125] Kimberly J Shillcutt. *Solar based navigation for robotic explorers*. Carnegie Mellon University, 2000.

[126] Alex Shum. Optimal direction-dependent path planning for autonomous vehicles. 2014.

[127] Alex Shum, Kirsten Morris, and Amir Khajepour. Direction-dependent optimal path planning for autonomous vehicles. *Robotics and Autonomous Systems*, 70:202–214, 2015.

[128] Mel Siegel. The sense-think-act paradigm revisited. In *1st International Workshop on Robotic Sensing, 2003. ROSE'03.*, pages 5–pp. IEEE, 2003.

[129] Nuno Silva, Richard Lancaster, and Jim Clemmet. Exomars rover vehicle mobility functional architecture and key design drivers. In *12th Symp. on Advanced Space Technologies in Robotics and Automation (ASTRA)*, 2013.

[130] Doug Sinclair and Jonathan Dyer. Radiation effects and cots parts in smallsats. 2013.

[131] Alberto Stabile, Valentino Liberali, and Cristiano Calligaro. Design of a rad-hard library of digital cells for space applications. In *2008 15th IEEE International Conference on Electronics, Circuits and Systems*, pages 149–152. IEEE, 2008.

[132] STMicroelectronics. Arm-based 32-bit mcu with 768 kb to 1 mb flash. `https://www.st.com/resource/en/datasheet/cd00253742.pdf`.

[133] BAE Systems. Bae systems current processors and single board computers. `https://www.baesystems.com/en-us/download-en-us/20190124214317/1434554723043.pdf`.

[134] BAE Systems. Rad5545 multi-core system-on-chip power architecture processor. `http://www.baesystems.com/en/download-en/20170525130030/1434571328901.pdf`.

[135] Vivienne Sze, Yu-Hsin Chen, Tien-Ju Yang, and Joel S Emer. Efficient processing of deep neural networks: A tutorial and survey. *Proceedings of the IEEE*, 105(12):2295–2329, 2017.

[136] Christian Szegedy, Sergey Ioffe, Vincent Vanhoucke, and Alexander A Alemi. Inception-v4, inception-resnet and the impact of residual connections on learning. In *Thirty-First AAAI Conference on Artificial Intelligence*, 2017.

[137] Keras team. Keras applications. `https://github.com/keras-team/keras-applications`.

[138] Vorago Technologies. Vorago producs. `https://www.voragotech.com/vorago-products`.

[139] Paul Tompkins, Anthony Stentz, and David Wettergreen. Mission-level path planning and re-planning for rover exploration. *Robotics and Autonomous Systems*, 54(2):174–183, 2006.

[140] Karen Ullrich, Edward Meeds, and Max Welling. Soft weight-sharing for neural network compression. *arXiv preprint arXiv:1702.04008*, 2017.

[141] Jorge Vago, Olivier Witasse, Pietro Baglioni, Albert Haldemann, Giacinto Gianfiglio, Thierry Blancquaert, Don McCoy, R ExoMars de Groot, et al. Esas next step in mars exploration. *ESA Bulletin Magazine*, 155:12–23, 2013.

[142] Jorge L Vago, Frances Westall, Andrew J Coates, Ralf Jaumann, Oleg Korablev, Valérie Ciarletti, Igor Mitrofanov, Jean-Luc Josset, Maria Cristina De Sanctis, Jean-Pierre Bibring, et al. Habitability on early mars and the search for biosignatures with the exomars rover. *Astrobiology*, 17(6-7):471–510, 2017.

[143] Asad Vakil, Jenny Liu, Peter Zulch, Erik Blasch, Robert Ewing, and Jia Li. Feature level sensor fusion for passive rf and eo information integration. In *2020 IEEE Aerospace Conference*, pages 1–9. IEEE, 2020.

[144] Todd L Veldhuizen. Five compilation models for c++ templates. In *First Workshop on C++ Template Programming*. Citeseer, 2000.

[145] Stylianos I Venieris and Christos-Savvas Bouganis. fpgaconvnet: A framework for mapping convolutional neural networks on fpgas. In *2016 IEEE 24th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 40–47. IEEE, 2016.

[146] Tanya Vladimirova, Xiaofeng Wu, and Christopher P Bridges. Development of a satellite sensor network for future space missions. In *2008 IEEE Aerospace Conference*, pages 1–10. IEEE, 2008.

[147] Dolores R. Wallace and Roger U. Fujii. Software verification and validation: an overview. *Ieee Software*, 6(3):10–17, 1989.

[148] Shiping Wang and Han Wang. Unsupervised feature selection via low-rank approximation and structure learning. *Knowledge-Based Systems*, 124:70–79, 2017.

[149] Christopher R Webster, Paul R Mahaffy, Sushil K Atreya, Gregory J Flesch, Michael A Mischna, Pierre-Yves Meslin, Kenneth A Farley, Pamela G Conrad, Lance E Christensen, Alexander A Pavlov, et al. Mars methane detection and variability at gale crater. *Science*, 347(6220):415–417, 2015.

[150] Minghan Wei and Volkan Isler. Building energy-cost maps from aerial images and ground robot measurements with semi-supervised deep learning. *IEEE Robotics and Automation Letters*, 5(4):5136–5142, 2020.

[151] Nathan Whitehead and Alex Fit-Florea. Precision & performance: Floating point and ieee 754 compliance for nvidia gpus. *rn (A+ B)*, 21(1):18749–19424, 2011.

[152] B Williams, P Antreasian, E Carranza, C Jackman, J Leonard, D Nelson, B Page, D Stanbridge, D Wibben, K Williams, et al. Osiris-rex flight dynamics and navigation design. *Space Science Reviews*, 214(4):69, 2018.

[153] Matthias Winter, C Barcaly, Vasco Pereira, Richard Lancaster, Marcel Caceres, NB McManamon, N Silva, D Lachat, and M Campana. Exomars rover vehicle: detailed description of the gnc system. *ASTRA*, 2015.

[154] Matthias Winter, Sergio Rubio, Richard Lancaster, Chris Barclay, Nuno Silva, Ben Nye, and Leonardo Bora. Detailed description of the high-level autonomy functionalities developed for the exomars rover. In *Proceedings of the 14th Symposium on Advanced Space Technologies in Robotics and Automation, Leiden*, pages 20–22, 2017.

[155] Markus Wulfmeier, Dushyant Rao, and Ingmar Posner. Incorporating human domain knowledge into large scale cost function learning. *arXiv preprint arXiv:1612.04318*, 2016.

[156] Markus Wulfmeier, Dushyant Rao, Dominic Zeng Wang, Peter Ondruska, and Ingmar Posner. Large-scale cost function learning for path planning using deep inverse reinforcement learning. *The International Journal of Robotics Research*, 36(10):1073–1087, 2017.

[157] Xilinx. Zynq-7000 soc product advantages. `https://www.xilinx.com/products/silicon-devices/soc/zynq-7000.html`.

[158] Guanjun Xu and Zhaohui Song. Effects of solar scintillation on deep space communications: challenges and prediction techniques. *IEEE Wireless Communications*, 26(2):10–16, 2019.

[159] Tomohiro Yamaguchi, Takanao Saiki, Satoshi Tanaka, Yuto Takei, Tatsuaki Okada, Tadateru Takahashi, and Yuichi Tsuda. Hayabusa2-ryugu proximity operation planning and landing site selection. *Acta Astronautica*, 151:217–227, 2018.

[160] Congrui Yi and Jian Huang. Semismooth newton coordinate descent algorithm for elastic-net penalized huber loss regression and quantile regression. *Journal of Computational and Graphical Statistics*, 26(3):547–557, 2017.

[161] Jure Zbontar and Yann LeCun. Computing the stereo matching cost with a convolutional neural network. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1592–1599, 2015.

[162] Chen Zhang, Peng Li, Guangyu Sun, Yijin Guan, Bingjun Xiao, and Jason Cong. Optimizing fpga-based accelerator design for deep convolutional neural networks. In *Proceedings of the 2015 ACM/SIGDA international symposium on field-programmable gate arrays*, pages 161–170, 2015.

[163] Kun Zhou, Xiangxi Meng, and Bo Cheng. Review of stereo matching algorithms based on deep learning. *Computational Intelligence and Neuroscience*, 2020, 2020.

[164] Chenzhuo Zhu, Song Han, Huizi Mao, and William J Dally. Trained ternary quantization. *arXiv preprint arXiv:1612.01064*, 2016.

[165] Barret Zoph, Vijay Vasudevan, Jonathon Shlens, and Quoc V Le. Learning transferable architectures for scalable image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 8697–8710, 2018.